Rapport de Stage de DEA
Étude de la Compilation du *High Performance Fortran*

Fabien Coelho
<coelho@cri.ensmp.fr>

été 1993

Étude de la Compilation du *High Performance Fortran*

Résumé

Le *High Performance Fortran Forum* a défini au cours de l'année 92 un nouveau standard pour la programmation des machines parallèles a hiérarchie mémoire. Ce langage, le *High Performance Fortran*, est basé sur Fortran 90 avec l'addition de directives de placement des données (`align`, `distribute`), des boucles parallèles (directive `independent`, construction `forall`) et quelques autres extensions. L'étude a porté sur la compilation des langages pour machines parallèles, et un prototype de compilateur pour le langage HPF a été réalisé. Ce prototype implémente la méthode de résolution dynamique des accès aux variables réparties, comme base de compilation systématique des programmes, et une analyse des recouvrements comme optimisation. Le langage source du prototype est basé sur Fortran 77 augmenté des directives statiques et de l'expression du parallélisme de HPF. La cible est Fortran 77 avec la librairie d'échange de messages PVM 3. Des mesures de performances des codes générés ont été réalisées sur réseau de stations de travail, avec un exemple de relaxation Jacobienne. Les résultats sont comparés aux performances obtenues par d'autres prototypes, comme Adaptor et Fortran D.

High Performance Fortran Compilation, Study and Implementation

Abstract

The High Performance Fortran Forum defined in 1992 a new language for distributed memory and non-uniform memory access multiprocessors. This language, called HPF, is based on Fortran 90 and a set of new directives and constructs used to express data distribution and parallelism. In this report, compilation techniques for multiprocessors are presented and a prototype compiler for the High Performance Fortran is described. The prototype implements the run-time resolution method, plus an overlap analysis for optimisation. The prototype source language is Fortran 77 with static data placement and parallel directives. The output language is Fortran 77 with calls to PVM 3-based run-time support functions. Performances of the generated code for Jacobi iterations have been measured on a network of workstations. They are compared with results obtained by other prototypes, such as Adaptor and Fortran-D.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Les besoins des ingénieurs et scientifiques en puissance de calcul augmentent aussi vite, sinon plus, que les puissances proposées par les supercalculateurs traditionnels. D'après une annonce de presse de Fujitsu pour son dernier supercalculateur parallèle, les problèmes auxquels s'attaquent maintenant les ingénieurs et scientifiques sont parmi les plus complexes de la planète, et ces défis sont connus sous le nom de *Grand Challenges*. La liste de ces problèmes a été établie à la fin des années 80 par un ensemble de spécialistes des différents domaines, chargés de faire de la prospective sur ce qui pourrait être fait avec des puissances de calcul formidables, telles qu'on peut les espérer avec les ordinateurs massivement parallèles.

1.1 *Grand Challenges*

Cette liste a permis de définir les objectifs des programmes de recherche aux États-Unis pour le domaine de l'informatique et la dernière décénnie du second millénaire.

On y trouve notamment :

- la modélisation globale de la planète, au plan climatologique, océanique, pour résoudre les questions de pollution et de dégradation de la couche d'ozone ;
- le développement de sources d'énergie alternatives au pétrole ;
- dans le domaine de la physique, la chromodynamique quantique et l'étude des semi-conducteurs et des supraconducteurs ;
- le projet génome humain, qui permettra de détecter et soigner les maladies génétiques ;
- en Intelligence Artificielle, les problèmes de vision et de cognition seraient de nature à utiliser ces supercalculateurs.

Un certain nombre de grands problèmes industriels sont prêts également à absorber les puissances proposées :

- la localisation des nappes de pétrole de manière plus efficace que le essai et erreur utilisé actuellement, grâce à des codes de calculs plus performants ;
- la simulation de réactions chimiques par les industries pharmaceutiques pour développer de manière plus rapide et moins chère de nouveaux médicaments ;

Sans compter les applications plus traditionnelles, mais grosses consommatrices de puissance (comptée en Megaflops, Gigaflops voire pour l'avenir Teraflops, *i.e.* millions, milliards voire milliers de milliards d'OPérations en virgule FLottante par Seconde), telles :

- la mécanique des fluides (automobile, aéronautique, etc.),
- la résistance des matériaux (calculs de structure...)
- etc.

De nombreux constructeurs internationaux tentent de satisfaire ces besoins, parmi lesquels Cray, Meiko, Thinking Machine, Fujitsu, IBM, et le constructeur français Archipel.

1.2 Le Calcul Scientifique Aujourd'hui

Les progrès et les gains de puissance des supercalculateurs ont été très importants ces vingt dernières années, notamment grâce à l'augmentation de la fréquence d'horloge des machines. Cela a été possible grâce à une miniaturisation toujours plus poussée et à la spécialisation des circuits [HP90]. Cependant la marge de manœuvre est maintenant très limitée : on est en effet arrivé aux limites du possible (des temps de quelques nanosecondes par cycle d'horloge par exemple), et la progression des performances à l'avenir ne pourra se faire de la même manière.

Pour assurer l'évolution en puissance nécessaire des matériels, de nouvelles voies ont été explorées [HP90], dont le principe fondamental est élégant de simplicité : puisqu'on ne peut pas faire les opérations plus vites les unes derrière les autres, il suffit que plusieurs opérations ou instructions aient lieu en même temps ! On cherche donc à faire du parallélisme, que ce soit au niveau d'un processeur, ou en faisant coopérer plusieurs processeurs sur un même problème. On n'échappe cependant pas à la séquentialité intrinsèque des algorithmes, qui borne les espoirs d'accélération que l'on peut nourrir, comme le montre la loi d'Amdalh [Amd67].

1.2.1 Parallélisme Monoprocesseur

Diverses techniques matérielles ont été développées dans ce sens [HP90], telles le *pipeline*, le *superscalaire* et le *vectoriel*:

- le **pipeline** : au niveau du microprocesseur, les opérations de base, comme les additions ou multiplications flottantes, prennent un certain temps qu'il est difficile de réduire ; l'idée est de découper ces tâches en opérations élémentaires (alignement des mantisses, addition, normalisation...), effectuées sur des portions de circuit isolées les unes des autres ; quand une des sous tâches est finie, la portion de circuit libérée peut être utilisée pour une autre opération ; chaque opération, individuellement, prend le même temps, mais plusieurs opérations peuvent être en route au même moment, à des étapes différentes du calcul ; si cette possibilité est exploitée par le compilateur, on augmente ainsi le débit d'opérations que fait le circuit ; l'avantage de cette technique est qu'elle permet d'augmenter le débit de calculs et d'opérations avec une quantité de circuiterie pratiquement constante, si les compilateurs exploitent ces possibilités ;
- le **superscalaire** : cette technique consiste à permettre au processeur d'exécuter plusieurs instructions de types différents par cycle d'horloge, dans la mesure ou ces instructions sont indépendantes les unes des autres ; c'est là encore le compilateur qui

doit se charger de générer un code à la fois correct et optimal, et cette phase de compilation est plus critique au niveau des boucles, où l'exécution passe le plus de temps [GS92, WE93] ;

- le **vectoriel** : cette dernière technique consiste simplement à multiplier la circuiterie utilisable par le processeur pour faire certaines opérations (typiquement, du calcul flottant), et de gérer les registres sous forme de vecteurs ; ainsi, le processeur peut lancer de la même manière des opérations qui seront exécutées sur des vecteurs.

Ces techniques ont amené les supercalculateurs sur la voie du parallélisme. Elles ne sont cependant pas suffisantes pour atteindre les puissances fantastiques visées par les constructeurs pour l'avenir, et qui s'expriment en *TeraFlops* (milliers de milliards d'instructions en virgule flottante par seconde !). Un parallélisme plus massif est nécessaire pour cela, mais il est également beaucoup plus difficile à mettre en œuvre.

1.2.2 Parallélisme Multiprocesseurs

L'idée du parallélisme sur plusieurs processeurs n'est pas récente. En effet, déjà dans les années soixante on construisait des machines composées de processeurs élémentaires reliés par un réseau d'interconnexion. Chaque processeur élémentaire était un petit ordinateur comportant une unité de traitement, de la mémoire, une interface vers le réseau et parfois des périphériques d'Entrées-Sorties [Wil93].

Cette idée est prometteuse car on peut envisager ainsi de repousser très loin les limites de ce qui peut être fait des puissances qui pourraient être atteintes et exploitées à l'avenir. Le principe est extensible (*scalable*), c'est à dire qu'il fonctionne encore quand on change d'échelle, contrairement aux autres techniques de gains de puissance comme le raccourcissement des temps de cycle des machines, qui ne peuvent être réduits indéfiniment.

Cependant sa mise en pratique pose quelques problèmes. En effet, faire marcher des centaines, voire des milliers de processeurs ensemble et de manière performante est loin d'être facile. Les différentes tâches doivent agir de manière coordonnée pour être efficaces. Des communications et des synchronisations sont nécessaires entre les processeurs.

Il existe différents types de machines parallèles multiprocesseurs, et le nombreuses taxonomies ont été proposées pour les classer, les désigner et les comparer[BD93, Fly66, Fly72, Joh82, RF76, Hän82]. La plus connue de ces taxonomies est certainement celle de Flynn [Fly66]. Les architectures parallèles modernes sont généralement classées d'après la taxonomie de Flynn, en ajoutant des considérations sur la manière dont est gérée la mémoire entre les processeurs (partagée ou répartie).

SM : machines à mémoire partagée (*shared memory*) :

il existe, physiquement ou virtuellement, une mémoire commune à tous les processeurs qui se partagent ainsi les données ; il faut cependant arriver à gérer les conflits d'accès et les problèmes de cache au niveau de cette mémoire, ce qui est déjà un problème pour les machines séquentielles ; la mémoire est considéré comme un goulot d'étranglement des performances sur les ordinateurs modernes ; commercialement, KSR propose des machines sur ce principe [OR93] ;

DM : machines à mémoire répartie (*distributed memory*) :

chaque processeur a sa propre mémoire, et il n'existe pas de mémoire globale pour la mise en commun des données ; l'accès à des données non locales doit donc se faire en accédant à la mémoire du processeur propriétaire, souvent au moyen de l'échange explicite de messages (*message passing*); l'avantage de cette architecture est qu'elle est dans son principe extensible, mais la difficulté de les programmer efficacement est un frein à leur développement.

SIMD : flot d'instruction unique, (*single instruction, multiple data*):

chaque processeur élémentaire effectue les même instructions sur ses données locales, au même moment. La CM–2 de *Thinking Machine* fonctionne sur ce principe ;

MIMD : flots multiples, (*multiple instruction, multiple data*):

les processeurs exécutent chacun leur programme indépendamment les uns des autres, et se synchronisent éventuellement en échangeant des messages.

1.3 Problèmes

Les problèmes auxquels doivent faire face les utilisateurs sont de trois natures: la programmation de ces machines, la portabilité des programmes développés entre machines différentes, et l'obtention de performances raisonnables, c'est à dire de l'ordre de grandeur des performances de crête de la machine.

1.3.1 Difficultés de Programmation

Les machines parallèles les plus prometteuses au plan performances et extensibilité, les machines à flot multiple d'instructions et de données, et à mémoire répartie (MIMD-DM) sont également les plus difficiles à programmer. La manière normale de ce faire est de programmer des processus échangeant des messages, et qui seront exécutés par différents nœuds de la machine. Mais ce modèle de programmation est très difficile à mettre en œuvre dans la pratique:

- le programmeur ne manipule pas les données directement, mais des données morcelées ; les erreurs sont donc fréquentes et difficiles à détecter ;
- le déboguage de ces programmes sur une machine parallèle est très difficile, car de nombreux programmes s'exécutent de manière concourante (et non plus un seul programme), et des bogues peuvent de ce fait apparaitrent de manière non déterministe ;
- quand aux performances, elle dépendent fortement de la manière dont sont placées et découpées les données sur les différents processeurs, et l'ajustement de ce placement oblige à réécrire des parties importantes du code ;

Au delà de ces difficultés propres aux machines à mémoire répartie, il reste le problème de la programmation parallèle, dont la correction doit être assurée. Les algorithmes séquentiels doivent souvent être revus pour en exploiter le parallélisme, pas toujours évident à détecter.

1.3.2 Portabilité Réduite

L'écriture de programmes pour les machines à mémoire répartie est un investissement, important compte tenu des difficultés de programmation de ces machines, et cette investissement est particulièrement risqué, car on n'est pas assuré de sa pérénnité:

- le constructeur peut disparaître et la machine ne plus être maintenue ;

- le modèle de programmation des machines d'un même constructeur peut évoluer, suivant les nouveaux choix architecturaux des nouvelles machines ;

Pour se garder de ces problèmes, des démarches de normalisation ont eu lieu ou bien encore sont en cours :

- **MPI** : *Message Passing Interface* ;
- **HPF Forum** : *High Performance Fortran Forum* ;
- **HPC Forum** : *High Performance C Forum* ;
- **PCF Fortran** : *Parallel Computing Forum Fortran.*

De plus, des librairies d'échange de messages ont été portées sur de nombreuses architectures, telles PVM, P4, Parmacs, Express...

1.3.3 Performances Décevantes

Les machines parallèles sont généralement basées sur une hiérarchie de mémoires [HP90], et le placement des données dans cette hiérarchie influe beaucoup sur les performances, de manière plus importante encore que sur les machines séquentielles. En conséquence, il est très courant de parvenir à des performances très médiocres par rapport à ce que devrait permettre la machine.

1.3.4 Conséquences

La conséquence de ces difficultés est que le développement d'applications parallèles revient très cher, et que la portabilité des programmes est réduite, ce qui ne pousse pas à l'investissement compte tenu des risques. Les constructeurs de machines, et la communauté du parallélisme en général se sont donc penchés sur ces problèmes, pour relâcher ce frein au développement qu'est le manque de clients...

L'objectifs de ces recherches est de proposer aux programmeurs un modèle de programmation convivial, le plus proche possible de ce qu'il peut rencontrer pour les machines séquentielles, tout en offrant à la fois une bonne portabilité et le moyen d'accéder aux performances maximales de la machine. Vœux pieux qui ont donné lieu à de nombreux développements, dans des voies diverses et souvent complémentaires.

Ce rapport présente d'abord, au chapitre 2, une étude bibliographique des différentes solutions qui ont été envisagées dans la littérature pour résoudre ces différents problèmes. L'une des solutions est de faire porter les difficultées de programmation des machines parallèles, spécifiquement des machines MIMD, sur le compilateur, des extensions de langage permettant d'aider le compilateur dans cette tâche. Le *High Performance Fortran*, langage standard défini par un Forum, est présenté au chapitre 3. L'étude formelle d'un schéma de compilation systématique est présenté ensuite, au chapitre 4. La réalisation d'un prototype de compilateur suivant le schéma proposé est décrite au chapitre 5. Enfin les mesures de performances des codes générés par le prototype son présentées et comparées à d'autres outils au chapitre 6, avant de conclure.

Chapitre 2

Étude Bibliographique

Cet chapitre présente les différentes voies explorées pour tirer le maximum de performances des machines parallèles présentes et futures. Ce problème est l'un des principaux points de difficultés qui doivent être surmontés par la communauté scientifique du parallélisme pour imposer les supercalculateurs parallèles dans le domaine du calcul scientifique. La difficulté principale de ces machines réside dans leur programmation *efficace*, et non pas dans des problèmes techniques matériels.

2.1 Automatisation

La première solution qui vient à l'esprit est de proposer le même modèle de programmation sur machines parallèles que sur machines séquentielles, ou peu s'en faut, en laissant reposer sur un autre niveau la distribution du code et le maintien de la cohérence, soit sur le matériel et le système d'exploitation, soit sur le compilateur.

2.1.1 Support matériel *(Virtual Shared Memory)*

Les machines parallèles à mémoire partagée sont beaucoup plus simple à programmer que les machines à mémoire répartie. L'idée est de proposer ce même environnement pour une machine à mémoire répartie, en donnant la charge de la gestion de la mémoire partagée au matériel et au système d'exploitation, qui se chargent donc du maintien de la cohérence et des communications nécessaires par un système de cache et pagination au niveau de la machine parallèle. Cette voie a été ouverte par Kai Li [Li86], et est explorée par le constructeur Kendall Square Research avec la KSR-1 [OR93, Ler93], et par le projet Koan [AP92].

2.1.2 Compilation Automatique

L'autre solution est de laisser le problème en entier à un compilateur optimisant et intelligent, qui devra assurer le portage complet du code initial et séquentiel sur une machine parallèle. Cette voie a été suivie, entre autres, par Hans Michael Gerndt dans sa thèse [Ger89] et John Merlin dans ses travaux sur Fortran 90 [Mer92]. Certains de ces travaux ont donné lieu à des prototypes. Un tel compilateur doit alors assurer les tâches suivantes :

- détection du parallélisme, si le langage ne l'exprime pas ;
- placement des données ;
- distribution du code *(owner compute rule, ldots)* ;

2.1.2.1 Vectorisation, Parallélisation Automatique

Les recherches dans le domaine de la parallélisation automatique ont connus des développements importants, principalement depuis l'arrivée sur le marché des supercalculateurs vectoriels, à la fin des années 70, où ce sujet, la vectorisation et la parallélisation automatique de programmes, est devenu d'une importance industrielle immédiate. Ce domaine de recherche est loin d'être clos, mais des prototypes d'un bon niveau existent, tels le programme PIPS [IJT91] (*Paralléliseur Interprocédural de Programmes Scientifiques*), ou des outils commerciaux comme Forge90 [Lev92].

La parallélisation automatique reste cependant une entreprise difficile, et ce pour plusieurs raisons. D'abords, la détection du parallélisme n'est pas toujours facile. Les applications séquentielles ne sont pas « pensées » en fonction de ces contraintes, et les algorithmes doivent souvent subir des transformations importantes pour en extraire le parallélisme. De plus, il arrive souvent que l'algorithme même doive être changé, pour un autre dont le parallélisme est plus accessible. Le choix des transformations de programme est difficile pour un paralléliseur automatique, et le changement d'algorithme est en général hors de portée, car les résultats peuvent être changés. La seule exception concerne des algorithmes aux propriétés très particulières comme les réductions.

2.1.2.2 Placement automatique de données

Une fois le parallélisme du programme détecté, encore faut-il décider du placement des données de manière à exploiter aux mieux ces parties parallèles et à minimiser les communications résultantes. Ce domaine est là encore un secteur actif de la recherche, et des revues des travaux récents sont proposées, par Ferrante *et al.* [FGMS92] ou Lefebvre [Lef93]. Divers travaux de recherche proposent des algorithmes de placement, en ayant éventuellement un autre objectif principal : Ancourt et Irigoin [AI92],Darte et Robert [DR93], Feautrier [Fea93], Knobe et Natarajan [KN90], Knobe, Lukas et Steele [KLSJ90], Anderson et Lam [AL93b] ou Hassaine [Has93].

Le choix du placement est éventuellement un non-choix si celui-ci est décidé de manière canonique, uniquement à partir de la déclarations des tableaux [AI92, KN90]. Il peut être basé sur une analyse plus ou moins fine de tout ou partie du programme.

Un algorithme de placement automatique comprend généralement :

- un « morceau » de programme, plus ou moins contraint (programme entier, nid de boucles uniforme [DR93], nid de boucles parallèles...) ;
- sur ce « morceau » de programme, un certain nombre d'analyses sont calculées et utilisées (vecteur de dépendance, profondeur de dépendance, information intra et/ou interprocédurales...) ;
- un modèle de coûts est choisi, et un objectif d'optimisation par rapport à cette fonction est donné (minimisation des volumes de communication et du temps global d'exécution sous ce modèle, etc.).

Les méthodes les plus générales, comme [DR93, Fea93], cherchent à la fois une fonction de placement pour les tableaux et une fonction de temps (*schedule*) pour les itérations des boucles. Le placement va donc de paire avec à la fois la distribution du code et l'exploitation du parallélisme, ce qui semble naturel.

Dans [KN90], un graphe de préférences est déterminé en fonction des alignements nécessaires aux différentes affectations du programme, et ce graphe de préférence est exploité pour déterminer un bon alignement.

2.1.2.3 Distribution Automatique

Ce dernier point est sans doute celui qui focalise le plus de recherches actuellement. Connaissant le parallélisme d'un programme et le placement des données sur une machine parallèle, il s'agit de générer un code le plus efficace possible pour l'exécution en parallèle. C'est soit une analyse intelligente, soit un utilisateur qui peut avoir indiqué au système de compilation les parties de programme parallèles et la répartition des données en mémoire.

Cette phase est indépendante des deux autres, mais un « bon » placement ne peut être décidé qu'avec une idée précise de ce qui peut être généré à cette phase, et les algorithmes de placement automatique font souvent partie d'une phase de distribution automatique.

De nombreux projets de recherche, voire prototypes de compilateurs ont été développés pour automatiser cette tâche. Certains produits commerciaux sont même proposés par des constructeurs. Ces travaux sont brièvement présentés en 2.3.

2.1.2.4 Conclusion

La parallélisation de programme et le placement de données automatiques ont donné de bons résultats, mais on est encore au début du développement industriel des méthodes proposées, et certain problèmes résistent et résisteront encore à l'automatisation. Cependant les utilisateurs souhaitent avoir un moyen de contrôler ce qui est fait par un compilateur, et pouvoir ajuster de manière plus intelligente le placement que ce que ne pourra jamais faire une procédure automatique.

Pour permettre d'attendre des compilateurs plus intelligents, et pour autoriser les utilisateurs à régler finement le parallélisme et le placement des données d'un programme, diverses extensions de langages on été imaginées, le compilateur n'ayant plus qu'à assurer la distribution du code le plus efficacement possible, en suivant les indications données. À la limite, ces extensions de langage peuvent être envisagées comme cible d'un paralléliseur-placeur automatique.

2.2 Langages

Un certain nombre de langages, nouveaux ou basés sur des langages existants, ont donc été définis afin de permettre à l'utilisateur de spécifier le placement et le parallélisme d'un programme. Certaines de ces extensions ont donné lieu à des prototypes de compilateurs. Ces extensions partent d'un langage de base généralement séquentiel et y ajoutent un moyen d'exprimer le placement des données souhaité, ainsi que des possibilités de décrire le parallélisme de l'application. La review de Crooks et Perrott [CP93] présente les différentes constructions proposées pour décrire le placement des données dans différents langages.

2.2.1 Langage de base

Les langages permettant la spécification du placement et du parallélisme sont souvent des extensions de langages existants, ce qui permet d'espérer ralier de nombreux utilisateurs de ces langages. Les langage de base les plus souvent choisis pour base d'une extension sont Fortran et C, mais d'autres projets s'appuient sur des sources différentes. Il est bon que noter que le langage de base est souvent réduit à ses aspects les moins exotiques , ce qui fait que la sémantique dont les compilateurs doivent tenir compte est souvent comparable.

2.2.1.1 Fortran

Le choix prépondérant de Fortran, que ce soit Fortran 77 [ANS83] ou Fortran 90 [ISO91], n'est pas un hasard :

- Fortran est le langage de choix du monde scientifique depuis son apparition en 1954 (Fortran I par Backus), et les tentatives de pousser ce langage hors jeu pour le remplacer par un autre [Can92] plus « moderne » et plus « propre » ne semblent pas aboutir rapidement ;
- les utilisateurs sont formés à ce langage ;
- de très nombreux programmes existent qu'il n'est pas envisageable de recoder, ce sont les « cartes poussiéreuses » (*dusty deck*) ;
- les optimisations des compilateurs pour ce langage sont plus poussées que pour d'autres langages, car la norme le permet, et la demande de compilateurs spécialement performants est forte de par la large utilisation des codes écrits en Fortran.

Fortran s'impose donc naturellement comme le langage de choix pour une extension destinée à utiliser des supercalculateurs à mémoire distribuée pour le calcul scientifique. C'est l'analyse qui a été faite par les constructeurs de ces machines [Pla92, Pas92, MPM92, Lim93], mais aussi par certains groupes de recherche et développement :

- *Rice University* [HKT92a, MFvL+92, HKT92b, Tse93] ;
- Université de Vienne [ZC93, BCZ92, ZBC+92, CMZ92] ;
- le *GMD* [Bra92d, Bra92c, Bra92a, Bra93b, Bra93a, Bra92b] ;
- le projet SUPERB [Ger89, GZ90, Ger91] ;
- ou encore [Mer92, CC92], ARF, etc.

2.2.1.2 Langage C

Le second choix de langage de base est sans conteste le langage C de Kernigan et Rithchie [KR78]. Les raisons de ce choix sont moins évidentes, du moins en ce qui concerne le monde scientifique qui continue à programmer en Fortran. Le langage C est néanmoins très largement utilisé par les programmeurs dans le monde, et son développement va de paire avec celui du système d'exploitation UNIX dont il est une des bases.

Le langage C est à la base des projets suivants : Octi [LPT92], Pandore [APT90, ACP92], DINO et Dataparallel C [HLQA92].

2.2.1.3 Autres Langages

D'autres langages sont également considérés comme base de départ pour des extensions, comme Modula-2 ou Modula-3 [PHL93, Hei93], Pascal pour le compilateur Kali [KMVR90, KM91]. D'autres travaux s'appuient sur des langages fonctionnels, comme *Id nouveau* [RP89, RP92], ou *Crystal* à Yale [LC91]. Enfin des projets utilisent un langage propre, c'est le cas pour *Booster* présenté aussi dans [CP93].

2.2.2 Extensions pour le Parallélisme

À partir de ces langages de base, les différents auteurs ont ajouté, quand nécessaire, un moyen de spécifier le parallélisme, et la manière de l'exploiter, pour une application. Pour les langages fonctionnels, une telle extension n'est pas nécessaire puisque le parallélisme est implicite : les fonctions n'ayant pas d'effets de bord, toutes les opérations qui n'utilisent pas les résultats l'une de l'autre sont indépendantes et peuvent être exécutées de manière concourante.

2.2.2.1 Boucles Parallèles

Les extensions permettant de spécifier le parallélisme dans une application se limitent pour les langages dédiés au calcul scientifique à diverses formes de boucles parallèles, dont l'objectif est d'exploiter le **parallélisme de données** des calculs, par opposition à au **parallélisme de tâches** (faire deux actions sans rapports de manière simultanée). Les extensions proposées sont adaptées au parallélisme de données, données qui sont à comprendre comme un ensemble de tableaux cartésiens. Dans cet esprit, diverses formes de boucles sont proposées :

- la boucle parallèle **indépendante**, qui spécifie que les itérations d'une boucles sont indépendantes les unes des autres ; on retrouve cette sémantique de boucle dans Vienna Fortran (boucle `forall`), mais aussi dans HPF [For93] (directive `independent`) ou encore dans les extensions de Modula [Hei93, PHL93] ; cette sémantique de boucle est plus contrainte que la sémantique de boucle data-parallèle ;

```
c independent loop in
c   High Performance Fortran
chpf$ independent
      do 10 i=1,n
 10     a(i) = a(i)+b(i)
```

```
(* independent loop in modula-2* *)
forall i:[1..n] in parallel
  a[i]:=a[i]+b[i]
end;
```

- la boucle **data-parallèle**, généralement baptisée `forall`, a une sémantique différente de la boucle indépendante : les valeurs lues pour une affectation sont les valeurs initiales des tableaux en entrée de boucle, et non celles éventuellement modifiées qui résulteraient de l'exécution séquentielle de la boucle ; cette sémantique de boucle est retenue par de nombreuses extensions, comme Fortran D [HKT92a, MFvL$^+$92, HKT92b, Tse93], mais aussi HPF [For93] (boucle `forall`), ou encore dans Modula-3* [Hei93] ;

  la sémantique data-parallèle est naturelle pour les machines SIMD, et apparaît comme allant de soi avec la notation des sections de tableaux de Fortran 90 [ISO91] ; le `forall` permet d'étendre l'utilisation de cette sémantique.

```
c array sections
c   in Fortran 90
c vector shift
      v(2:n) = v(1:n-1)
c matrix sum
      a(:,:) = b(:,:)+c(:,:)
```

```
c forall in
c   High Performance Fortran
c matrix transpose
      forall(i=1:n, j=1:n)
     &  a(i,j)=a(j,i)
```

Au delà de ces simples boucles, certaines extensions permettent des ajustement complémentaires :

- **masquage** : on peut dans certains langages ajouter un prédicat à une boucle *data-parallèle*, qui sélectionne une partie du domaine pour faire l'affectation commandée ;

c'est le cas de la boucle `forall` de HPF [For93], du `where` de Fortran 90 [ISO91].

```
c masked forall in High Performance Fortran
      forall(i=1:n, j=1:m, y(i,j).ne.0.0)
     &   x(i,j) = 1./y(i,j)


c Fortran 90 where example,
c   a is an array
      where (a>0.0) a = 1.0/a
```

- **localisation** : le langage peut permettre également à l'utilisateur de spécifier sur quels processeurs doivent être faits les calculs d'une boucle parallèle, au moyen d'une clause « `on` », où l'on précise la localisation, généralement en désignant le propriétaire d'un élément de tableau ; on trouve de telles directive dans le compilateur Kali [KMVR90, KM91], dans Fortran D [HKT92a, MFvL+92, HKT92b, Tse93], dans le langage ARF [CP93], dans Vienna Fortran [ZC93, BCZ92, ZBC+92, CMZ92], etc.

```
c Fortran D on clause
      forall i=1,n on home(a(i))
        b(i+5) = a(i)+b(i)+c(i)+d(i)
      endfor
```

2.2.2.2 Réductions, Autres Extensions

Les opérations de réduction sont une forme différente de parallélisme exploitable par une machine à mémoire distribuée. Elles sont plus dures à repérer et exploiter, et sont donc mises en évidence par certains langages qui proposent des moyens de les désigner dans le programme, et de laisser aux soins du compilateur et des fonctions de support leur exploitation. Les réductions apparaissent fréquemment dans les codes de calcul, et de nombreuses extensions les comprennent, à la manière du standard Fortran 90 [ISO91] qui les définit comme fonctions de base du langage (*fonctions intrinsèques*).

Ces réductions peuvent porter :

- sur le calcul d'une valeur réduite par un prédicat (*max, min...*) ;

```
c reduction in Fortran 90
      x = maxval(a)
c x is the maximum value of the elements of array a(...)
```

- sur le calcul d'une valeur réduite par une opération (*somme de termes...*) ;

```
c reduction in Vienna Fortran
      forall i=1,n
        reduce(add, x, a(i))
      end forall
c x is the sum of a(1:n)
```

– sur la localisation d'une valeur réduite par un prédicat (*localisation du maximum...*).

```
c location reduction in Fortran D
      do i=1,n
        reduce(max, a(i), idx, i)
      enddo
c idx is such that a(idx) is the max of a(1:n)
```

Le standard Fortran 90 [ISO91] autorise également des masques pour les opérations de réduction, ce qui permet d'écrire :

```
c masked reduction in Fortran 90
      x = sum(a, mask=a>0)
c x is the sum of all positive elements of array a(...)
```

2.2.3 Extensions pour le Placement

Il est apparu comme critique de permettre à l'utilisateur de décrire la manière dont il veut voir placées les différentes données du programme. Les extensions de langage portent leur attention sur la description du placement souhaité pour des tableaux cartésiens, ce qui correspond aux besoins du monde des applications scientifiques, puisque cette partie des programmeurs s'est toujours contentée du langage Fortran 77 [ANS83], qui n'autorise que cette structure de données !

Les différentes extensions de langage pour ce propos sont décrites dans la review de Crooks et Perrott [CP93], qui présente en détail SUPERB, Id nouveau, Kali, ARF, ADAPT, Vienna Fortran, Fortran D, Booster, et plus brièvement Pandore II, Adaptor, DINO, Dataparallel C, Blaze, Baber's Hypertasking system, Linda, Crystal, Paragon, AL, MIMDizer et ETHZ Oxygen.

De ces nombreux exemples on peut tirer quatres aspects qu'il semble important de maitrîser pour décrire le placement des données (tableaux cartésiens) souhaité pour une application :

– les **distributions régulières**, qui permettent de décrire un placement « régulier » du tableau sur les processeurs ;

– les **distributions irrégulières**, pour les placements « irréguliers » ;

– les **alignements**, qui autorisent à décrire la distribution d'un tableau relativement à un autre tableau, qui sert de référence, et qui a éventuellement une existence virtuelle ;

– les modifications **dynamiques** de placement, au cours de l'exécution du programme, entre deux phases n'ayant pas les mêmes besoins.

Au delà de ces aspects, on peut noter que la machine parallèle cible est souvent vue comme un tableau cartésien de processeurs, comme dans Kali, Vienna Fortran, Fortran D, ADAPT, etc. C'est également le cas pour Adaptor, même si cette vue est implicite.

De plus, certaines des extensions proposées ajoutent des fonctions permettant de s'enquérir du placement courant des données, permettant ainsi d'orienter le déroulement de l'exécution en fonction du placement et de la configuration disponible. C'est notamment le cas de Vienna Fortran (`select dcase`), et du *High Performance Fortran* (`number_of_processors()`, `hpf_alignment()`...).

2.2.3.1 Distributions Régulières

Toutes les extensions de langage proposées possèdent généralement des directives permettant de découper le tableau par *block* ou de manière *cyclic*, et de répartir les morceaux obtenus sur un tableau cartésien de processeurs. Par exemple :

- Kali [KMVR90, KM91] ;
- Vienna Fortran [ZC93, BCZ92, ZBC+92, CMZ92] ;
- Fortran D [HKT92a, MFvL+92, HKT92b, Tse93] ;
- Adaptor [Bra92d, Bra92c, Bra92a, Bra93b, Bra93a, Bra92b] ;
- Pandore [APT90, ACP92] ;
- des diverses extensions constructeurs de Fortran [Pla92, Pas92, MPM92] ;
- etc.

Cette façon de procéder a été reprise par le *High Performance Fortran* [For93].

Les avantages de cette manière de procéder sont multiples, ce qui explique le consensus autour de ces types de placement :

- ces placements résument ce qui est souhaitable pour permettre de bons découpages pour la plupart des algorithmes courants dont les accès se font par voisinage ;
- il est facile de décrire les données locales à différents processeurs par des formules simples et efficaces, lorsque les calculs effectués sont simples et réguliers ;
- lorsque les accès sont dans un voisinage, la génération des messages est facilitée ;
- le rôle des distributions cycliques est de permettre un équilibrage de charge que n'est pas exprimable avec des distributions par *block*, lorsque par exemple un algorithme n'accède qu'à une partie d'un tableau (matrice triangulaire, etc.).
- le langage *High Performance Fortran* [For93] généralise les distributions *block* et *cyclic* en permettant de distribuer les dimensions en les découpant par *block*, puis en liant ces blocks à un nombre moindre de processeurs de manière cyclique.

Il est à noter que certaines extensions sont limitées, c'est à dire que l'on ne peut pas distribuer arbitrairement les dimensions des tableaux de données manipulés par l'application, et que certaines restrictions doivent être appliquées, par exemple :

- Adaptor n'autorise la distribution que de la dernière dimension du tableau, par *block*, ou le placement sur le processeur *host* ;

```
c mapping of arrays with ADAPTOR
c which uses CM Fortran like directives
      real a(n,n), b(n,n)
cmf$  layout a(:news) ! a is block distributed on the last dimension
cmf$  layout b(:host) ! b is mapped on processor host
```

- le prototype de compilateur de Fortran D [Tse93] n'autorise que la distribution d'une dimension, même si le langage en permet plus ;

De telles restrictions simplifient également d'un bon ordre de grandeur la complexité du compilateur qui doit gérer les multiples cas possibles de distribution. De telles directives de placement ne permettent que des placement réguliers, mais il arrive que certaines application aient besoin d'une plus grande latitude pour décrire un placement efficace (blocks de taille irrégulière, duplication partielle des données, etc.).

2.2.3.2 Distributions Irrégulières

Des placements plus exotiques de données sont parfois nécessaires à certaines applications, et certaines extensions de langage permettent de spécifier ces exotismes :

- **duplication** : les tableaux dont les valeurs sont souvent utilisées et rarement définies doivent pouvoir être placés avec un certaine dose de duplication des valeurs ; Vienna Fortran, Fortran D et HPF permettent ce genre de manipulations ;

```
c High Performance Fortran
c replication of vector v on processors p(:)
      real v(n)
chpf$ template t(n)
chpf$ align v(*) with t(*)
c every processors will have a copy of vector v
```

- **irrégularités régulières** : par un jeu sur les directives d'alignement et distribution du *High Performance Fortran*, on peut également découper un tableau de manière « pseudo irrégulière » ; dans l'exemple suivant, les processeurs ont successivement 1 et 2 éléments de tableaux :

```
c High Performance Fortran
c regular irregular distribution
      real v(n)
chpf$ template t(2*n)
chpf$ align v(i) with t(2*i-1)
chpf$ distribute t(block(3))
c this gives 1 then 2... elements to successive processors
```

  de même la méthode de placement préconisée par le projet Octi [LPT92] autorise des placements étranges qui apparaissent par des jeux entre un premier niveau de placement « à trous », et un second niveau qui place les éléments résultants sur les processeurs, sans suivre les dimensions initiales ;

- **placement irréguliers** : Vienna Fortran ainsi que Fortran D autorisent enfin l'utilisateur à définir ses propres fonctions de placement, ce qui permet de traiter les indirections (matrices creuses, éléments finis), la fonction devant mettre directement en correspondance un élément de tableau avec le numéro du processeur sur lequel il doit être mis :

```
c irregular distribution in Fortran D
c map is replicated on every processors, and defined somewhere
      integer map(n), a(n)
```

```
            decomposition irreg(n)
            align a with irreg
            distribute irreg(map)
```

Il est bien de permettre à l'utilisateur toutes les fantaisies dans le placement des données, mais cela est fait au prix de la performances de l'application, car il est difficile de compiler efficacement des codes utilisant de tels placements [MSS+88].

2.2.3.3 Alignements

Les différentes sortes de distributions qui ont été décrites précédemment permettent de préciser le placement souhaitable pour la plupart des applications courantes, qui ne nécessitent que des distribution par blocks ou bien cycliques. On pourrait donc penser que ces méthodes de placement sont suffisantes...

Cependant la distribution indépendamment les uns des autres des tableaux de données ne fait pas apparaître les liaisons induites sur les différents éléments par les accès se faisant aux mêmes itérations. Des tableaux de données différents doivent être maintenus en correspondance, et donc par exemple être distribués de la même façon.

De plus, dans le processus d'optimisation d'une application, le programmeur est amené à essayer différents placements pour les données de l'application, mais ces distributions ne sont pas indépendantes les unes des autres, compte tenu des correspondances à maintenir. On voudrait donc pouvoir préciser une seule distribution pour l'application, la distribution de chaque tableau étant dérivée à partir d'une mise en correspondance des tableaux les uns par rapport aux autres...

De ces remarques de bon sens a été tiré le concept d'alignement, qui permet de préciser quelles doivent être les dispositions relatives des tableaux de données les uns par rapport aux autres, indépendamment de leur distribution finale. Cette idée a été exploitée dans Vienna Fortran, Fortran D, ainsi que dans le langage intermédiaire utilisé pour la compilation de Modula-3* [Hei93], et a été reprise lors de la définition du langage HPF [For93].

Fortran D et HPF utilisent de manière facultative un espace intermédiaire (*decomposition, template*) sur lequel sont faits les alignements, et qui est indépendant des tableaux de données du programme, ce qui permet de disposer d'un espace plus grand qu'aucun des tableaux de l'application.

Pratiquement, les alignements sont précisés à l'aide d'expressions affines de variables muettes liants alors les dimensions de deux tableaux alignés, ou éventuellement de sections de tableau ayant la même signification.

```
c an alignment in High Performance Fortran
chpf$ template t(3*n,n,n,n)
chpf$ align a(i,j,k,*) with t(2*i+n-5,k,n+1-j,*)
chpf$ align b(:,:,:,*) with t(n-3:3*n-5:2,n:1:-1,:,*)
```

Fortran D permet de plus des précisions sur la manière de gérer un dépassement de l'alignement sur une dimension, à l'aide du complément de directive `overflow` :

```
c overflow use in Fortran D
      real a(n,n,n)
      decomposition d(n, n, n)
      align a(i, j, k) with d(i-1, j, k+1) overflow(trunc, error, wrap)
```

Fortran D permet également de préciser qu'une partie seulement d'un tableau doit être aligné, à l'aide du complément `range` qui permet de décrire les sections du tableau initial auxquelles s'appliquent l'alignement. On peut penser que ces dernières extensions sont un peu inutiles et, qui plus est, difficiles à gérer par le compilateur.

2.2.3.4 Placement Dynamique

Pour de très nombreuses applications, il n'y a pas de placement, alignement et distribution qui soit optimal d'un bout à l'autre de l'exécution. Le placement optimal évolue au contraire selon les différentes phases de calcul, d'ou l'idée de pouvoir le modifier de manière dynamique.

Michel Loi dans [Loi92] présente ses travaux concernant la redistribution dans le cas des arrangements de processeurs hypercubiques, et on trouve dans [KLD92] des considérations sur les optimisations possible lors des réalignements.

Vienna Fortran permet de redistribuer des tableaux de manière dynamique avec la commande `distribute` (par opposition à l'annotation `dist` pour le placement statique), qui est suivie de la liste des tableaux impliqués ainsi que de la spécification de la nouvelle distribution. Seuls les tableaux qualifiés de `dynamic` peuvent être redistribués.

Fortran D autorise à la fois le réalignement et la redistribution des données, et les directives de placement `align` et `distribute` peuvent être utilisées statiquement ou dynamiquement. Fortran D impose que les placements dynamiques soient connus de manière statique, au moment de la compilation.

Le *High Performance Fortran* [For93] reprend ces traits avec les directives statiques `align` et `distribute`, et les directives exécutables `realign` et `redistribute`, pour les tableaux déclarés `dynamic`.

```
c High Performance Fortran realign and redistribute
      real, dimension(n):: a,b,c
chpf$ dynamic a,b
chpf$ align a with b
chpf$ distribute b(block)
chpf$ distribute c(cyclic)
      ...
chpf$ realign a with c
chpf$ redistribute b(cyclic)
c this example doesn't make much sense...
```

2.3 Distribution Automatique

À partir d'un langage précisant le placement des données et le parallélisme de l'application, d'une manière intuitive pour les programmeurs, il faut maintenant que le compilateur puisse générer du code pour une machine à mémoire répartie fonctionnant par échange de messages. Cette approche de la compilation est décrite dans un article de Callahan et Kennedy [CK88b], en insistant sur le fait que l'essentiel de la difficulté porte sur le compilateur, qui doit générer un code par échange de messages efficace.

2.3.1 Contraintes et Choix

Un certain nombre de contraintes doivent être prises en compte, et de choix faits pour mettre en pratique cette démarche, assurer la généralité des solutions proposées et simplifier la complexité du compilateur. Les chemins suivis par la plupart des projets dans ce domaine, et leur justification, sont présentez ci-après :

2.3.1.1 Modèle d'exécution SPMD ou 2PMD

Un modèle d'exécution du programme généré par le compilateur doit être choisi. Pratiquement tous les projets s'appuient sur un modèle **SPMD** (*Single Program, Multiple Data*) ou bien **2PMD** (*2 Programs, Multiple Data*). Ce choix s'impose de lui même, et seul le projet de Rogers et Pingali [RP89, RP92] basé sur le langage fonctionnel *Id nouveau* propose de sortir de ce cadre en particularisant plus finement le programme généré aux différents processeurs qui devront l'exécuter.

Le Modèle **SPMD** consiste à exécuter le même programme sur les différents processeurs de la machine parallèle, l'exécution du programme étant paramétrée par l'identitée du processeur. Pour être efficace et raisonnable, un programme SPMD ne doit pas être trop particularisé (même déclarations locales des données, etc.). Ce choix s'impose pour plusieurs raisons :

- le procédé est *extensible*, c'est à dire ne dépend pas directement du nombre de processeurs de la machine cible (dans l'hypothèse inverse, une machine à 1000 nœuds nécessite la génération et la compilation de 1000 programmes différents !) ;
- le modèle de programmation à parallélisme de données se prête bien à ce type de compilation, les processeurs exécutant *grosso modo* les mêmes calculs sur leurs données locales, et les communications étant généralement sous forme de motifs répétés sur la plupart des processeurs ;

Sur beaucoup de supercalculateurs parallèles modernes [Ler93, OR93], les nœuds de la machine sont homogènes, sauf un seul particularisé qui se charge notamment des entrées-sorties, généralement baptisé *host*, par opposition aux *nodes*. Il est alors naturel de générer un programme particulier pour ce processeur, celui-ci ne devant généralement pas participer aux calculs. C'est de cette façon qu'il faut comprendre le modèle **2PMD**, un programme pour le processeur *host*, et un pour les *nodes*.

Ce modèle est choisi par de nombreux projets, comme Vienna Fortran, Adaptor, le prototype de compilateur Fortran 90 pour machine à mémoire répartie de Chen et Cowie [CC92], le compilateur Octi [LPT92], etc. Le prototype de compilateur de Fortran D [Tse93, HKT92a] n'a pas pris cette voie car les entrées-sorties ne sont pas traitées.

2.3.1.2 *Owner Rule*

Une autre décision est souvent prise lors de la conception des prototypes de compilateur, pour définir quel processeur doit effectivement assurer le calcul d'une itération donnée. Le choix se porte sur presque toujours sur le propriétaire de la variable définie par l'affectation contenant les calculs. Ce choix n'est pourtant pas optimal, mais il facilite la gestion du compilateur. Cette règle de décision porte le nom de *owner rule*.

Seuls les langages proposant une clause « `on` » (Fortran D, Kali, ARF, Vienna Fortran, etc.) permettent d'échapper à cette règle de par la volonté de l'utilisateur. On peut supposer que des compilateurs « intelligents » pourront un jour répondre de manière rationnelle à cette question. Le langage HPF ne permet par à l'utilisateur de spécifier la localisation des calculs.

Il est à noter que l'exploitation de cette règle n'est pas évidente, dans le cas des tableaux répliqués, où un choix particulier doit être fait : soit on applique la règle et les calculs sont faits plusieurs fois, sur différents processeurs ; soit un des détenteurs fait les calculs et envoie une copie de la valeur aux autres détenteurs de la variable. Le choix optimal n'est pas simple, et dépend du nombre de variables redéfinies, du nombre de copies et des performances du réseau de communication de la machine.

2.3.1.3 Obligations

Avec un langage où le placement et le parallélisme est spécifié, un compilateur réaliste a deux obligations :

- **tout compiler** : le compilateur doit accepter tous les codes légaux en entrée, et les compiler de manière à garantir la correction de l'exécution du programme ; la solution *obligée* s'appelle la résolution dynamique, et est présentée à la section 2.3.2 ;
- **optimiser** : la méthode de résolution dynamique présentée ci-après est particulièrement inefficace, et pour prétendre améliorer le temps d'exécution par rapport à une exécution séquentielle du programme, il est nécessaire de procéder à un minimum d'optimisations (voir section 2.4).

2.3.2 Résolution Dynamique

La nécessité de compiler tous les programmes, même ceux sur lesquels les optimisations ne peuvent être appliquée, oblige au développement d'une méthode de compilation systématique, dont le principe est le suivant :

- chaque nœud parcours tout le flot de contrôle du programme ;
- à chaque opération, il détermine dynamiquement quelle doit être sa participation :
  - aucune participation ;
  - exécuter le calcul ;
  - envoyer un donnée au nœud devant exécuter l'opération ;
  - etc.

Cette méthode est appelée la résolution dynamique des accès aux variables réparties (*run-time resolution*), parce qu'on détermine au moment de l'exécution la localisation effective des données auxquels une opération doit accéder. Elle est présentée par Callahan et Kennedy [CK88b], par Rogers et Pingali [RP89, RP92] et dans le projet SUPERB par Zima et Gerndt [Ger89]. Tous les projets présentent cette méthode à la base de leur processus de compilation, comme Pandore [APT90, ACP92], Adaptor, etc.

Le fonctionnement de cette méthode oblige le programme exécuté sur les nœuds de la machine parallèle à se poser dynamiquement de nombreuses et questions sur la localité des données, l'identité des autres processeurs, etc. Pour y répondre, il faut que le système ait prévu :

- des fonctions de support répondant aux interrogations des nœuds ;
- une description de placement des données sur les différents processeurs ;
- des routines de communication « point à point », qui permettent d'envoyer une donnée nécessaire à n'importe quel processeur.

Cette méthodologie a l'avantage de ne nécessiter aucune information au moment de la compilation, pour pouvoir exécuter n'importe quel programme. Elle est facile à implémenter, puisqu'il suffit de garder chaque accès à une variable répartie par les vérifications et les communications éventuellement nécessaires, et de garder les calculs pour le processeur devant les exécuter. Elle est présentée comme une simple réécriture du code initial à la section 4.2, pour générer deux codes (modèle 2PMD).

Cependant la résolution dynamique paye sa généralité au prix de la performance :

- chaque processeur exécute tout le flot de contrôle du programme initial (c'est à dire qu'il parcourt toutes les opérations du programme) ;
- il doit s'interroger sur sa participation éventuelle à ces opérations (exécution de calcul, envoie et/ou réception de messages...) ;
- répondre à ces questions nécessite de nombreuses opérations, particulièrement dans le cas de placement un peu complexes, comme c'est le cas dans le *High Performance Fortran* ;
- même pour une boucle parallèle, les communications peuvent séquentialiser l'exécution ;

Pour les applications courantes, les calculs des opérations élémentaires nécessitent un temps négligeable par rapport aux multiples traitements nécessaires pour répondre aux questions et aux temps de communication interprocesseurs. Le seul espoir de gagner du temps avec une telle distribution est que tous les accès soient locaux, et que le temps de calcul des opérations soit très important. Si des communications doivent avoir lieu, le programme risque d'être séquentialisé par les attentes de messages [CK88b].

L'inefficacité de la résolution dynamique oblige donc naturellement à envisager une politique d'optimisation aggressive, qui doit être gérée par le compilateur de manière automatique, et en assurant la correction des opérations faites.

2.4 Optimisations

L'inefficacité de la résolution dynamique des accès aux variables réparties oblige à une politique d'optimisations essentielle pour espérer accéder à un niveau de performances correctes avec une distribution automatique du code. Si les compilateurs ne parviennent pas à un assez bon niveau de performances, les difficultés de programmation des machines parallèles resteront un frein à leur développement...

On trouve dans la thèse de Tseng [Tse93] une description des nombreuses optimisations proposées pour le prototype de compilateur de Fortran D, et dans Zima et Chapman [ZC93] une description des optimisations envisagées pour le prototype de compilateur du langage Vienna Fortran.

2.4.1 Bases

Les optimisations présentées ci-après concernent les problèmes particuliers posés par la distribution de code automatique. On ne présente donc pas les optimisations en aval de cette phase, que ce soient les optimisations pour programmes séquentiels, pour lesquels on pourra se référer au *Dragon book* de Aho, Sethi et Ullman [ASU86], ou les optimisations concernant le parallélisme intra-processeur (utilisation des *pipelines*, exploitation des processeurs superscalaires [GS92, WE93], etc.).

2.4.1.1 Mémoire

Il est nécessaire d'ajuster les déclarations locales des processeurs, en suivant les indications de placement. Des contraintes contradictoires apparaissent :

- minimiser l'espace mémoire alloué ;
- permettre un accès aléatoire en $O(1)$;

Dans la description du placement du langage HPF (section 3 de [For93]), les distributions provoquant des « irrégularités régulières » présentées plus haut 2.2.3.2, provoquent un placement irrégulier dont la minimisation de l'allocation mémoire se fait en perdant la complexité « instantanée » de l'accès aléatoire. C'est par exemple le cas de la démarche adoptée par Chatterjee *et al.* [CGL+93] et par Gupta *et al.* [GKM+93].

2.4.1.2 Résolution Dynamique

Si suffisamment d'informations statiques sur le placement sont disponibles, quelques améliorations sont possibles sur le code généré pour la résolution dynamique.

Placement Similaire : Dans le compilateur Octi [LPT92], les auteurs proposent de factoriser les gardes lorsque le placement et l'accès à deux données sont similaires. Il est alors également facile de regrouper les messages de données de placement « similaire », et de gagner ainsi le temps de latence dû à l'envoi de plusieurs messages.

Particularisation : Rogers et Pingali [RP89, RP92] particularisent les codes générés au processeur devant les exécuter. Ce procédé est envisageable pour des machines de quelques processeurs, ou si seulement quelques particularisations sont faites pour des types de processeurs (bords, intérieur d'un domaine...), mais n'est pas envisageable à grande échelle. Pour savoir si la particularisation des codes générés vaut la peine, il faudrait mesurer le poids des gardes dans ces codes.

2.4.1.3 *Inspector Executor*

L'un des problèmes principaux de la résolution dynamique est que les messages sont échangés référence par référence, ce qui conduit le programme à une très grande inefficacité : non seulement chaque nœud exécute le flot de contrôle de la machine dans son entier, mais de plus chaque message échangé est atomique, et le programme est encore ralenti des temps de latence des messages.

La technique de l'*Inspector-Executor* présentée dans le cadre du projet Kali [KMVR90, KM91] consiste à, dans une première phase *Inspector*, déterminer pour un nid de boucles parallèles les ensembles de données qui doivent être échangés avec les autres processeurs et éventuellement les itérations à calculer localement, puis dans une seconde phase *executor*, à échanger ces messages et faire les calculs locaux nécessaires. Cette méthode à quelques avantages :

- factorisation des messages ;
- pas de séquentialisation de l'exécution dûe aux échanges de messages ;
- bonne rentabilité si le « motif » de message est répété sur un grand nombre d'itérations, et que ce fait est détecté statiquement : on exécute alors une seule fois l'*Inspector* et de nombreuses fois l'*Executor*.
- cette technique permet de gérer des situations pour lesquelles une bonne compilation n'est pas envisageable (placement irréguliers...), et les ensembles peuvent éventuellement être déterminés au moment de la compilation [ESH93].

Cependant sa mise en pratique n'est pas simple :

- poids de la gestion des ensembles de données à échanger ;
- le flot de contrôle est parcouru plusieurs fois si on ne détermine pas les itérations à effectuer localement.

Cette technique permet dans certains cas une exécution moins inefficace, mais elle ne doit être employée que si les techniques d'optimisation statiques ne peuvent aboutir.

2.4.1.4 Informations Statiques

Les optimisations statiques, c'est à dire réalisées au moment de la compilation, nécessitent un certain nombre d'informations plus ou moins précises sur le programme, et qui peuvent être obtenues du programme de différentes manières :

- les analyses de programme, essentiellement les différents graphes de dépendances (chaînes de définitions-utilisations, profondeurs de dépendances, vecteurs de dépendances, etc.), mais aussi des analyses de flot de contrôle, propagation de constante, etc. [ZC90, IJT91]. Des analyses très précises de flot de données sont envisagées par Amarasinghe et Lam [AL93a] pour les optimisations de communications ;
- la directive `assert` du langage Vienna Fortran [ZBC$^+$92] permet au programmeur d'exprimer des assertions sur le programme ; à charge ensuite au compilateur d'utiliser ces informations.

En plus de ces informations tirées du programme, le placement des données de l'application peut avoir certaines propriétés connues ou détectables (inversibilité du placement des éléments de tableaux sur les processeurs, directions de réplication...) exploitables par le compilateur pour produire un meilleur code.

2.4.2 Messages et Itérations

Le fait que chaque nœud parcourt l'ensemble du flot de contrôle du programme, et que les messages soient envoyés référence par référence, sont les deux principales raisons de l'inefficacité de la résolution dynamique des accès aux variables réparties. Pour améliorer cette situation à la compilation, les premiers objectifs proposés par Callahan et Kennedy [CK88b] ont été :

- réduire l'ensemble des itérations auxquels doivent participer chaque nœuds autant que possible ;
- regrouper les envois de messages ayant même source et même destination ;

Dans de nombreuses situations ces deux tâches sont facile à appréhender de manière statique, et plusieurs méthodes ont été développées à ces fins. On trouve dans l'exposé de Pazat [Paz93] un très bref exposé de ces méthodes.

L'idée générale est de faire une analyse des recouvrements [Ger89] (*overlap analysis*) pour un nid de boucles, entre les données auxquelles accèdent les processeurs et les données disponibles sur les processeurs voisins, de manière à déterminer le contenu des messages à échanger et la taille des recouvrements à déclarer pour recevoir ces messages à leurs places relatives. Pour les boucles non parallèles, on doit également déterminer le placement du messages qui respecte les dépendances [Tse93]. Pour les boucles parallèles, les messages peuvent simplement être placés hors boucle. Cette méthode a été présentée dans le cadre du projet Superb, par Gerndt et Zima [Ger89, GZ90].

2.4.2.1 Méthodes Développées

Trois méthodes aux précisions différentes permettent de réduire les itérations locales de processeurs et de factoriser les messages à échanger. Elles sont de plus en plus lourdes, mais traitent des cas de plus en plus généraux. La finalité de ces méthodes est :

- de déterminer un sur-ensemble le plus petit possible des données devant être échangées, du point de vue de chaque processeur ;
- et de même pour les itérations devant être exécutées localement par chaque processeur (des gardes devant éventuellement filtrer les itérations à effectuer réellement).

Ces trois méthodes sont :

Méthode des Rectangles :

La description des ensembles est basée sur des rectangles, ce qui conduit dans certain cas à la définition de gros sur-ensembles pour les messages qui doivent être échangés. Cette technique est utilisée dans Superb [Ger89].

Descripteurs de Sections Régulières (*Regular Section Descriptors, RSD*) :

Les RSD ont été introduits par Callahan et Kennedy [CK88a] et Balasundaram et Kennedy [BK89], avec un objectif d'analyse de programme en vue d'une parallélisation automatique, et une implémentation est présentée dans [HK91]. Au lieu de considérer des sections d'éléments contigus des tableaux, on s'appuie sur une description à base de sections régulières :

- des points à égale distance les uns des autres, et représentés par un triplet d'entiers (premier point, dernier, et distance entre deux points).

– une version simplifiée de la méthode d'approximation par polyèdres limitée aux contraintes diagonales et parallèles aux axes, ce qui simplifie de manière significative les algorithmes.

C'est la technique de description, utilisée dans les travaux de Chau-Wen Tseng [Tse93] pour déterminer et manipuler les messages à échanger.

Polyèdres Convexes :

Cette dernière méthode est la plus précise de toutes, mais également la plus lourde. Les différents domaines sont approchés par des polyèdres convexes, sur les représentations desquels de nombreuses opérations peuvent être faites [Anc91]. Ces descriptions, et les manipulations induites, doivent faire appel à des algorithmes dont la complexité en temps et en espace est exponentielle. La résolution des système conduit parfois à dépasser les capacités limites des machines utilisées. Son application pour résoudre un cas très général de communications et de calculs pour le langage HPF est présentés dans [ACIK93].

Ces méthodes permettent donc de déterminer plus ou moins finement les données qui doivent être échangées et les processeurs qui sont impliqués par ces échanges de messages. Cependant, sauf pour la dernière méthode, il existe des cas pour lesquels ces analyses ne sont pas possible, particulièrement si la distribution des données n'est pas strictement par *blocks*, mais *cyclics* paramétré.

2.4.2.2 Cas Difficiles

Les directives de placement du langage HPF sont beaucoup plus générales que ce qui peut être traité directement par ces méthodes, et plusieurs auteurs se sont penchés sur le problème de la génération du parcours et des messages à échanger dans le cas le plus général de placement, lorsque le parcours est limité à des sections régulières des données.

Dans l'article de Chatterjee *et al.* [CGL+93], il est proposé d'utiliser des automates pour parcourir les données locales liées aux calculs à exécuter. En effet, le « motif » de parcours peut être généré et représenté pour chaque processeur. Un second niveau d'algorithme permet de traiter tous les cas de distributions et alignements. L'algorithme de détermination de l'automate doit être exécuté par chaque processeur. Pour les distributions « pseudo-irrégulières », on perd par une allocation optimale l'accès aléatoire instantané.

Gupta *et al.* [GKM+93], ainsi que Stichnoth [Sti93] présentent une approche basée sur des sections régulières, qui doivent être là aussi calculées pour chaque processeur. Dans [GKM+93], le cas le plus général est géré par une virtualisation des processeurs. La complexité des algorithmes est comparable à celle de la méthode présentée précédemment.

Dans [ACIK93], un système aux polyèdres est utilisé, et la détermination des différents ensembles fait appel à des algorithmes de complexité difficile [Anc91], qui doivent être déterminés statiquement.

Différentes approches ont donc été envisagées pour résoudre le problème des itérations locales et des données à communiquer dans le cas général du placement autorisé par les directives du *High Performance Fortran* [For93].

2.4.2.3 Optimisations des Messages

Une fois les messages déterminés, d'autres manipulations sont envisageables pour améliorer les performances. Dans le prototype de compilateur Fortran D [Tse93], les messages sont combinés pour diminuer le nombre de messages échangés et la quantité de données transmises :

- **coalescence** : les messages portant sur les mêmes données sont combinés de manière à ne pas envoyer plusieurs fois les mêmes données entre deux processeurs ;

- **aggrégation** : les messages ayant même source et même destination sont combinés en un seul message.

Un autre objectif est de reconnaître certains types de motifs de communication pour lesquels l'architecture est spécialement efficace moyennant un appel a des routines spéciales (*shift, all-to-all, broadcasts*, transposition, etc.). Li et Chen [LC91] proposent une technique basée sur une reconnaissance syntaxique de ces motifs de communications, à travers un filtrage (*pattern matching*). Cette technique est reprise pour Vienna Fortran [ZC93] et Fortran D [Tse93].

2.4.3 Transformations

Après avoir obtenus le programme et les messages à échanger, des transformations du programme sont envisageables qui visent à améliorer l'exécution du programme en essayant de superposer, quand c'est possible, les communications et les calculs. D'autres formes de transformations de programme permettent d'exploiter le parallélisme d'une boucle non parallèle, dont les dépendances se manifestent entre les processeurs (`doacross` de Cytron [Cyt86]).

2.4.3.1 Amont

Pour extraire le parallélisme de boucles présentant des dépendances, la technique des bases de temps, ou méthode hyperplane, a été proposée par Karp *et al.* [KMW67] et Lamport [Lam74]. Elle est développée dans Feautrier [Fea92a, Fea92b], qui a introduit le terme « base de temps », et étendue pour le placement dans Darte [DR93]. Elle consiste à trouver un ordre d'exécution des itérations d'un nid de boucle qui satisfasse les dépendances, qui soit linéaire, et qui minimise par exemple le temps d'exécution. Cette base de temps est représentée par un vecteur τ, et on programme en t les itérations i telles que $\tau.i = t$. Il reste un travail important pour exploiter de telles techniques dans le cadre de la distribution de code pour un langage comme HPF : il faut faire le lien entre les données sur les différents processeurs, les itérations programmées par la fonction de temps et les messages nécessaires.

Pour les boucles non parallèles de type `doacross` [Cyt86], Hiranandani, Kennedy et Tseng [HKT92a, HKT92b] cherchent à exploiter leur parallélisme de manière plus simple. La ou les boucles occasionnant des dépendances entre processeurs sont détectées. Le nid de boucle est réarrangée de manière à exploiter le parallélisme en *pipant* les messages avec la technique présentée ci-après. Un optimum doit être cherché et découpant la boucle par *blocks* pour trouver la meilleur taille qui équilibre délais de communications et calculs avec le temps de latences des messages.

Une autre approche est d'exploiter le parallélisme présent au niveau des instructions générées par le compilateur, pour exploiter efficacement des processeurs dont l'architecture est superscalaire ou VLIW. Ces techniques sont présentées dans Rau et Fisher [RF93], et le développement de nouvelles techniques continu [GS92, WE93].

La *Owner Rule* peut être également relachée, de manière à diminuer le nombre et la taille des messages à échanger : Le processeur un possède le plus de données localement se charge du calcul, et le résultat et ensuite stockés sur le « bon » processeur par un ultime message. Cette technique est utilisée par le projet Kali par exemple. Le problème se pose différemment pour la compilation des langages permettant de préciser la localisation des calculs, comme Fortran D ou Vienna Fortran : le compilateur doit alors suive les indications données, et ce n'est pas à une phase d'optimisation de prendre la décision de cette approche.

2.4.3.2 Pipe

Pour recouvrir les communications et les calculs, Rogers et Pingali [RP89, RP92] ont proposé de ramener vers le lieu de production l'envoi des valeurs des données nécessaires, tout en laissant leur réception au niveau de leur utilisation.

Cependant l'application de cette technique rentre en concurrence avec la vectorisation des messages, car au plus près du lieu de production les valeurs sont produites de manière atomique. Pour éviter ce problème, Tseng [Tse93] propose de rapprocher les messages de leur lieu de production après leur vectorisation. Il cherche ainsi à conserver les avantages des deux optimisations.

La recherche de l'optimum doit s'appuyer sur un modèle de la machine, et la meilleure solution peut être intermédiaire, par exemple en ajustant la taille des messages vectorisés.

2.4.3.3 Recouvrement des calculs et communications

Si la technique précédente ne peut être employée, il est par contre possible de réordonner les calculs et communications de manière à dissimuler les temps de communications par des calculs. L'ordre naturel est alors le suivant :

- envoi des messages ;
- calculs sur les données locales ;
- réception des messages ;
- calculs des itérations nécessitant des données non locales.

On essaie alors de dissimuler les temps de communication des messages par les calculs sur les données locales.

2.4.4 Autres Optimisations

D'autres optimisations ont été imaginées dans la littérature pour améliorer les codes générés dans les phases de distribution automatique.

2.4.4.1 Aval

Une fois les codes avec échanges de messages générés, des transformations peuvent encore être appliquées. Ces transformations nécessitent éventuellement des analyses particulières [Ger92], car le programme n'est pas à considérer comme un programme séquentiel mais plutôt comme un ensemble de programmes qui seront exécutés en parallèle, avec un paramétrage différent (modèle SPMD). Des transformations peuvent donc induire des blocages (*dead locks*)....

Pour la plupart des codes, une phase de distribution automatique qui aboutit à un programme SPMD devant être exécutés sur les nœuds d'une machine parallèle produit de très nombreuses gardes devant les routines de communication notamment, mais aussi pour certaines phases de calcul. Il est possible de factoriser et combiner ces gardes de manière à réduire le nombre de tests que devra faire chaque processeur. On peut également cherche à simplifier statiquement les gardes [Ger91].

Un moyen de réduire les communications est d'éliminer purement et simplement les messages inutiles, soit que la valeur précédemment reçue soit toujours valable, soit que le calcul d'une donnée soit inutile pour la suite des calculs [GGM93].

Dans le cas des programmes 2PMD, les transformations et éliminations de messages doivent être exécutées en parallèle sur les deux programmes, pour respecter la symétrie nécessaire entre envoi et réception de messages. Ce point est discuté ultérieurement dans le présent rapport.

2.4.4.2 Langages Fonctionnels

Le problème de l'utilisation des langage fonctionnels tels Sisal pour le calcul scientifique est leur compilation efficace [Can92] :

- il faut éviter de réallouer les tableaux de données à chaque redéfinitions des tableaux ;
- il faut reconnaître et stocker les invariants du programme pour ne pas les calculer de multiples fois.

Pour cela, des techniques d'optimisations spécifiques doivent être mises en places.

2.4.4.3 Autres Problèmes

D'autres problèmes existent, qui n'ont pas été traités ici :

- assurer une répartition de charge homogène aux différents processeurs (dans tous les cas où les processeurs pour lesquels le programme est compilé sont virtualisés) ;
- compiler les appels de procédures « distribués » de manière efficace ;
- etc.

2.5 Conclusion

De nombreux prototypes ont été implémentés qui exploitent ces idées, et les résultats obtenus [KF93] laissent à penser que dans de nombreux cas la programmation par échange de message peut être remplacée par celle à parallélisme de données. De nombreuses applications tests ont été développées pour évaluer les compilateur existant et à venir [BBLS91, BBDS93, MFvL$^+$92]. Cependant les prototypes ne sont pas encore à un niveau industriel, et la compilation efficace et complète d'un langage comme le *High Performance Fortran* [For93] est une entreprise difficile. Cependant on peut penser que beaucoup de problèmes sont à portée de la recherche actuelle, et que de bons outils seront à terme disponibles.

Chapitre 3

Présentation du *High Performance Fortran*

Le langage HPF (*High Performance Fortran*) a été défini au cours de l'année 1992 par un Forum, le HPFF. Ce Forum a défini le langage au moyen de votes successifs au cours de réunions qui ont eu lieu aux États-Unis. Le langage s'inspire de diverses extensions de Fortran pour machines à mémoire distribuée qui avait été proposé entre 88 et 91. Il permet par un jeu de directives de décrire le placement des données, et ainsi d'aider le compilateur à générer un code réparti.

3.1 Le Forum

Ce Forum a regroupé les principaux constructeurs, laboratoires et universités travaillant sur les machines à mémoire non uniforme (NUMA, *Non Uniform Memory Access*), c'est à dire les machines pour lesquelles le coût d'accès à une donnée peut varier selon la position de cette donnée dans les différentes mémoires. Il s'est formé et réuni pour mettre au point un nouveau langage de programmation, le HPF (*High Performance Fortran*), basé sur Fortran, et qui permette d'exploiter les potentialités de ces machines en tenant compte de l'état de l'art de la compilation, tout en offrant une programmation aisée et intuitive pour les utilisateurs.

On trouve notamment comme participants à ce forum :

- des **constructeurs** (de MMP ou de supercalculateurs) : Thinking Machine Corporation, Convex, IBM, DEC, Sun nCube, Intel SSD, ou encore l'entreprise française Archipel.

- des **universités** : Rice, Vienne, Syracuse, Yale, Cornell.

- des **laboratoires gouvernementaux** : Icase, LANL, LLNL, RIACS, GMD.

Cette participation particulièrement large, notamment en ce qui concerne les constructeurs, montre que le HPF est particulièrement important pour eux. HPF leur permettra de garantir aux utilisateurs la portabilité de leurs codes et la pérennité de leurs investissements. L'un des freins au développement commercial des ordinateurs à architecture parallèle disparaîtrait donc.

3.1.1 Buts du forum

Le Forum s'était donné un certain nombre de buts déclarés, dont l'accumulation peut laisser craindre que cela ne reste qu'une liste de vœux pieux. L'histoire de l'informatique dira si les choix faits auront été les bons...

- extension de Fortran pour la programmation parallèle, avec des performances optimales sur les machines MIMD et SIMD ;
- définition aussi proche que possible des autres standards, tels que Fortran-77 et Fortran-90 ;
- ouverture aux autres styles (*message-passing*) et langages de programmation ;
- proposition de critères de validation des compilateurs (*tests suite*, *i.e.* une suite de programmes de test pour comparer les performances des compilateurs HPF) ;
- disponibilité rapide des compilateurs (début 1993 !) ;
- simplicité maximale !
- large participation de la communauté, large distribution de la définition du langage ;
- présentation de la proposition dès Novembre 1992 (*Supercomputing'92*) ;
- possibilités d'évolution dans le cadre de la recherche.

Cependant il est évident que les différents participants n'ont pas les mêmes motivations pour participer au Forum, et on peut essayer d'expliciter les intérêts de chacun dans l'affaire :

- les constructeurs qui ne produisent pas de supercalculateurs surveillent ce qui se passe.
- les constructeurs de supercalculateurs veillent à ce que le langage défini permette d'exploiter les spécificités de leurs machines et, de préférence, pas beaucoup plus ;
- les différentes équipes de recherche présentes ont tendance à tirer la proposition vers les travaux qu'elles ont déjà faits, sans doute pour garantir à leur équipe une avance certaine et pour être parmi les premières à proposer un prototype de compilateur HPF, et parce qu'elles pensent proposer les meilleures solutions.

Toutes ces motivations ne sont heureusement pas divergentes !

3.1.2 Fonctionnement du forum

Le fonctionnement du Forum est assez intéressant et mérite de s'y arrêter. Les réunions ont eu lieu sur l'année 92 et le début de l'année 93, et ont abouti à une spécification de langage, qui doit faire office de standard pour la programmation des machines à hiérarchie mémoire, comme par exemple les machines à mémoire répartie.

3.1.2.1 Réunions et votes

Le Forum s'est réuni presque tous les mois aux États-Unis. Chacun des organismes a participé par l'intermédiaire d'un ou deux représentants. Les droits de vote étaient attribués aux organisations, et un représentant n'était autorisé à voter que s'il avait été présent à au moins deux des trois dernières réunions, la réunion courante étant comptée. Toute équipe ou entreprise intéressée pouvait envoyer un représentant aux réunions : le Forum était ouvert.

Les minutes des réunions du Forum ont été largement distribuées au moyen d'une liste de courrier électronique à laquelle tout le monde pouvait s'inscrire, sur simple requête. Plus de 550 alias figuraient sur cette liste, ce qui témoigne d'un intére certain au plan mondial, même si 80% de ces alias sont pointés sur les États-Unis. L'essentiel des personnes inscrites travaillent dans la recherche et il y a eu peu d'utilisateurs à proprement parler dans le forum. Sur cette liste étaient également envoyées les informations générales.

Lors des premières réunions du Forum, il est apparu que les réunions plénières n'étaient pas propices aux grandes discussions. En conséquence, des sous-groupes thématiques ont été formés. Ils présentaient le résultat de leurs réflexions internes en assemblée générale, où étaient prises les décisions.

3.1.2.2 Sous-groupes et listes électroniques

Voici la liste et le rôle des différents sous-groupes. Chacun d'eux possédait une liste de diffusion électronique sur laquelle les idées ou questions pouvaient être échangées. Des réunions de sous-groupes ont eu lieu avant les grandes réunions pleinières.

Distribute Ce sous-groupe se penche sur les directives de placement des données sur les processeurs, ainsi que sur les conventions d'appel des sous-programmes et procédures liées à la distribution.

F90 Ce sous-groupe a sélectionné ce que le *High Performance Fortran* devait retenir du standard Fortran 90.

Forall Ce sous-groupe s'est consacré à la spécification des boucles parallèles, dont une est justement appelée `forall`.

Intrinsics Ce sous-groupe a défini les nouveaux intrinsics du langage.

IO Ce sous-groupe a étudié les possibilités d'Entrées-Sorties parallèles.

Europe Ce dernier sous-groupe a été formé après coup, et a réuni des participants dont le point commun était d'être européens, et de vouloir maintenir le modèle de programmation par échange de messages qui a été financé depuis dix ans par la CEE.

Un sous-ensemble du langage a également été défini pour faciliter l'apparition des premiers compilateurs.

3.2 *High Performance Fortran*

Le Forum n'est pas parti de rien. Un certain nombre de recherches ont été effectuées sur le sujet, et des compilateurs basés sur des principes approchants sont déjà disponibles, soit sous forme de prototypes [BCZ92, CMZ92, ZBC+92, HKT92a, HKT92b, MFvL+92, Tse93], soit sous forme d'outils proposés par certains constructeurs [Pla92, MPM92].

Le choix de Fortran apparaît comme une nécessité, dans la mesure ou c'est le langage le plus utilisé par les applications scientifiques. Il ne serait pas imaginable de proposer autre chose et d'espérer un quelconque succès, vu la quantité de logiciels existants, de programmeurs formés, et de l'adéquation du langage au besoins (des boucles, des tableaux, des opérations). Contrairement aux matériels, les évolutions des techniques logicielles sont très lentes à passer dans les mœurs et les pratiques des programmeurs.

Fortran 90 [ISO91, CC92]

Fortran 90 est le nouveau Standard Fortran, dont la spécification est disponible sous la forme du document `ISO/IEC 1539 : 1991 (E)`. Par rapport à Fortran 77, ont été ajoutées de nouvelles constructions, qui permettent l'utilisation de l'allocation dynamique de la mémoire, les appels récursifs de fonctions et sous-programmes, ainsi que la possibilité de mieux encapsuler les différentes parties d'un programme. Du côté parallélisme, Fortran 90 permet de spécifier que des calculs ont lieu de manière parallèle sur des éléments de tableaux au moyen des *array sections* (par exemple : `a(1:N) = b(1:N) + c(1:N)`).

Cependant l'intérêt de Fortran 90 dans son entier n'est pas démontré. Le DoD (*Department of Defense, USA*) a rejeté en 1991 Fortran 90 pour ce qui concerne ses contrats, et n'a retenu comme langages que Fortran 77 et ADA.

CM-Fortran [Pla92] Il s'agit du Fortran proposé pour la Connection Machine. L'objectif est de pouvoir exploiter simplement le caractère SIMD de la CM-2. Par rapport à Fortran 77, le langage utilise les sections de tableaux de Fortran 90, un `forall` ainsi qu'un certain nombre de directives qui permettent de spécifier la manière dont doivent être placées les données.

Fortran-D [HKT92a, HKT92b, MFvL+92, Tse93] Il s'agit d'un langage, extension de Fortran, développé à l'université de Rice (Texas, USA), par le groupe de Ken Kennedy. Le D de Fortran-D n'est pas spécifié, mais peut être pris parmi les mots Directives, Données, Décomposition et Distribution. Il permet à l'utilisateur de diriger le compilateur quant à l'exploitation du parallélisme. L'utilisateur indique de quelle façon les données doivent se correspondre, ainsi que la manière dont elles doivent être distribuées entre les processeurs.

Vienna Fortran [BCZ92, CMZ92, ZBC+92, ZC93] Cette extension de Fortran est proposée par Hans Zima de l'université de Vienne. Comme Fortran-D, il permet de préciser le placement de données sur les processeurs, et l'utilisation d'un espace d'adressage uniforme comme pour la programmation de machines à mémoire partagée. Le compilateur doit se charger de l'insertion des messages.

MPP Fortran [Pla92, MPM92] Cette extension là est disponible sur les machines Cray.

PCF-Fortran Comme HPF, PCF-Fortran est le résultat d'un essai de standardisation par un Forum, le Parallel Computing Forum. Il est essentiellement destiné aux machines à mémoire partagée. Un `forall` y est également proposé.

3.2.1 Modèle HPF

Deux facteurs importants interviennent pour produire des codes performants sur les ordinateurs à architecture parallèles et mémoire répartie :

- l'exploitation du parallélisme des calculs ;
- les communications nécessaires pour cette exploitation.

HPF permet d'ajuster les programmes :

- en utilisant des constructions parallèles, telles que les boucles (parallèles, à parallélisme de données) ou les réductions, pour aider le compilateur à trouver le parallélisme présent dans le code dont l'exploitation sera la plus profitable ;
- en proposant des directives de placement des données sur les différents processus de la machine cible, qui permettent d'ajuster le placement de manière à minimiser les communications engendrées.

Par contre le Forum n'a pas fait de choix architecturaux dans la mesure du possible, et le langage doit pouvoir convenir aussi bien à des machines dont l'architecture est SIMD que MIMD, que la mémoire soit partagée ou répartie.

3.2.2 Survol de HPF

Le Forum a retenu dans la version finale de la spécification du langage HPF parue début mai 93 :

- Fortran 90 comme base de départ pour HPF [1] ;
- un ensemble de directives pour décrire le placement des données au moyen d'une spécification à 3 niveaux, et également des boucles parallèles.

L'idée des directives, qui sont sous forme de commentaires spéciaux, était initialement de ne pas modifier le programme initial et de n'ajouter que des annotations pour le compilateur. De cette façon, le code n'étant pas touché, rien ne s'oppose à ce qu'il soit compilé par un compilateur non HPF. Un principe de compatibilité *descendante* donc.

Ces directives commencent par les préfixes `CHPF$`, `!HPF$` ou `*HPF$`. Les directives peuvent être scindées sur plusieurs lignes, avec une syntaxe qui consiste à ajouter un esperluète (`&`) comme dernier caractère des lignes non terminales.

Cependant le principe des simples directives a été abandonné par le Forum, qui a défini de nouvelles instructions ainsi que de nouvelles fonctions intrinsèques. En conséquences, un programme HPF qui utilisera ces traits du langage perdra l'avantage de la compatibilité descendante.

1. Il est à noter que cette décision a d'importantes conséquences, car Fortran 90 est déjà un langage de taille importante, et d'une grande complexité : ainsi, trois ans après la parution de sa spécification, il n'existait pas encore de compilateur le compilant entièrement et efficacement

3.2.3 Placement des données en HPF

le modèle de placement des données de HPF comprends **3** niveaux pour mettre en correspondances les éléments des tableaux avec les processeurs physiques (figure 3.1).

alignment : les éléments de tableaux sont d'abords alignés sur un objet intermédiaire baptisé *template*.

distribution : les éléments de *template* sont ensuite distribués sur des *processors*, et tous les éléments de tableau alignés sur eux au passage.

mapping : les éléments de *processors* peuvent être considérés comme des processus, et ils doivent ensuite être placés sur les processeurs réels de la machine cible.

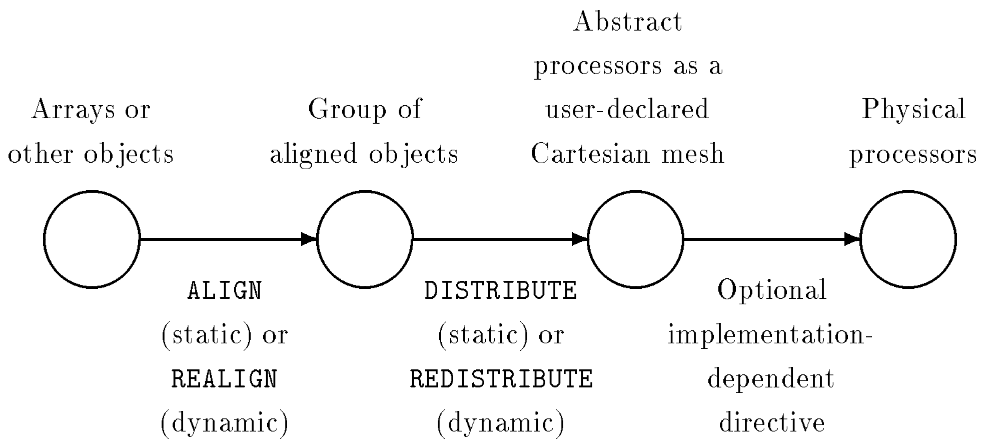

FIG. 3.1 - *Modèle de Placement de données de HPF*

Cette partie de HPF est décrite à la section 3.3.

3.2.4 Boucles Parallèles en HPF

HPF propose 3 sémantiques de boucle différentes :

- la boucle séquentielle `DO` (`DO` de la figure 3.2) ;
- les boucles à parallélisme de données `FORALL` (`FORALL` de la figure 3.2) ;
- la directive de boucle parallèle `INDEPENDENT` (figure 3.3).

Le principe de la boucle à parallélisme de données (*dataparallel loop*) est de spécifier directement l'ensemble des opérations à faire sur un tableau pris dans son entier. Cette sémantique est déjà présente sous forme implicite en Fortran 90, par l'utilisation de la notation dite des sections de tableau *array sections*. Ainsi une somme de matrices :

```
REAL A(N,N), B(N,N), C(N,N)
A(:,:) = B(:,:) + C(:,:)
```

ou même :

```
A = B + C
```

```
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```

```
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

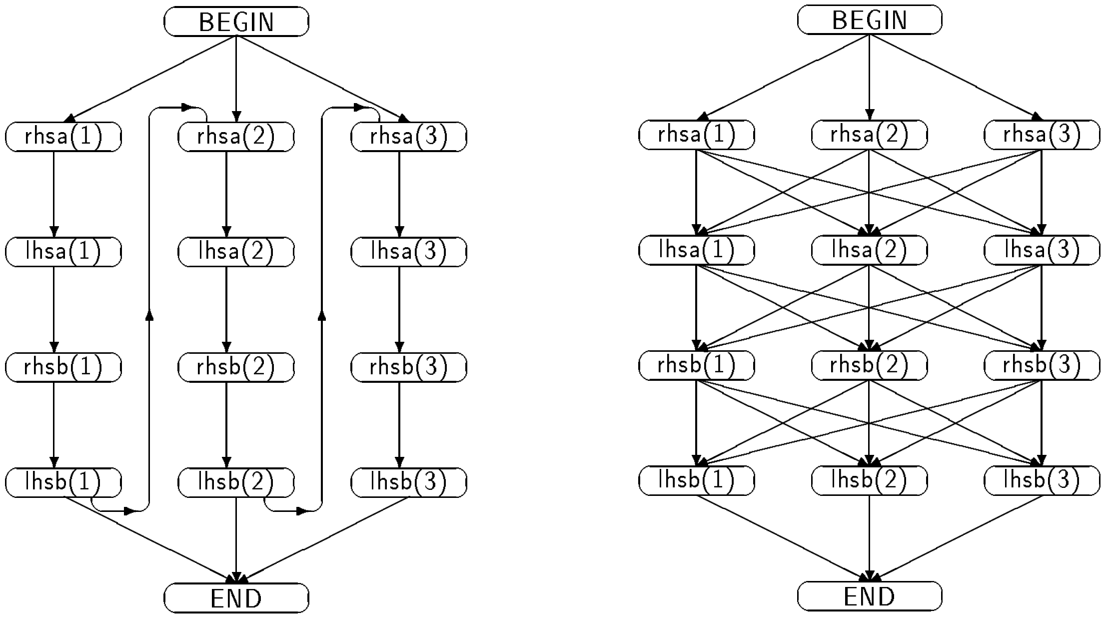

FIG. 3.2 - *Dépendances de `DO` and `FORALL` sans `INDEPENDENT`*

```
!HPF$ INDEPENDENT
DO i = 1, 3
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END DO
```

```
!HPF$ INDEPENDENT
FORALL ( i = 1:3 )
  lhsa(i) = rhsa(i)
  lhsb(i) = rhsb(i)
END FORALL
```

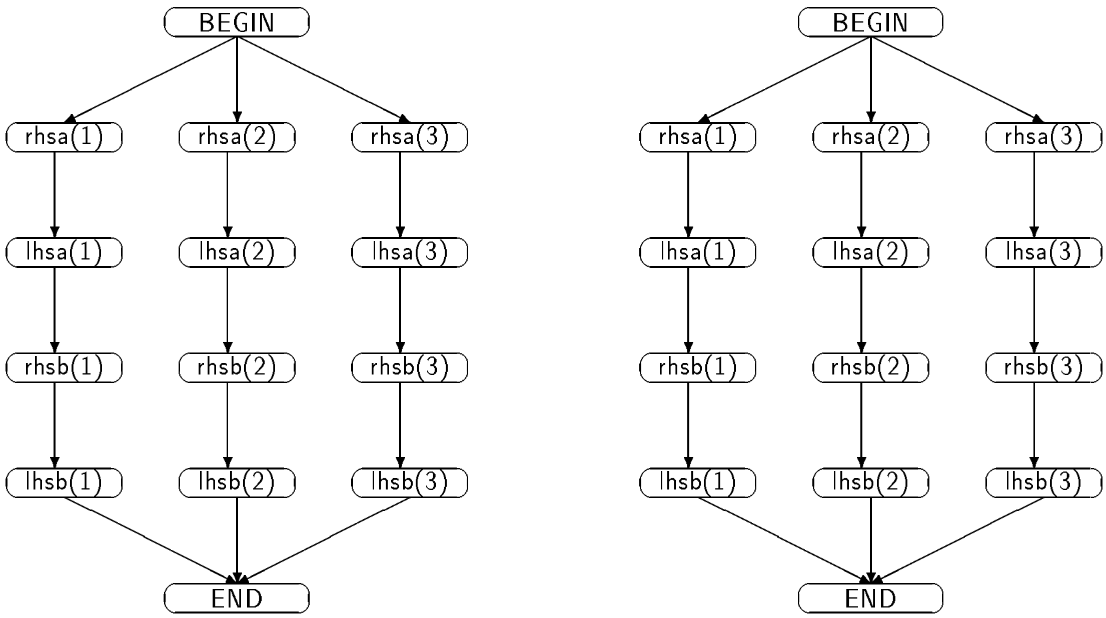

FIG. 3.3 - *Dépendances de `DO` et `FORALL` avec `INDEPENDENT`*

et le décalage à droite d'un vecteur (*shift*) :

```
D(2:21) = D(1:20)
```

Cependant cette notation est inutilement limitée par sa syntaxe, et on ne peut, par exemple, exprimer en l'utilisant une simple transposition. La boucle `FORALL` dans ses versions instruction et construction, complète donc cette lacune et étendant cette syntaxe en explicitant les indices de parcours des dimensions.

L'**instruction** `FORALL` généralise donc la notation des sections de tableau de Fortran 90, avec la même sémantique. Elle ne permet qu'une seule affectation, et a un masque optionnel :

```
FORALL (i=1:n, j=1:n) x(i,j) = y(j,i)

FORALL (i=1:n, y(i).NE.0.0) x(i) = 1.0 / y(i)

FORALL (i=2:n-1)
  x(i) = 0.5*x(i) + 0.25*(x(i-1)+x(i+1))

FORALL (i=1:n) x(i) = SQRT(y(i))
```

La **construction** `FORALL` étend l'instruction en permettant plusieurs affectations, ainsi que des constructions plus complexes à l'intérieur du corps de boucle, comme un `WHERE` ou un autre `FORALL`.
Exemples, initialisation d'une matrice carrée à l'identité :

```
FORALL(i=1:n, j=1:n)
  WHERE (i.EQ.j)
    a(i,j) = 1.0  ! diagonale
  ELSEWHERE
    a(i,j) = 0.0  ! reste
  END WHERE
END FORALL
```

ou affectations multiples, avec appel à des fonctions intrinsèques :

```
FORALL(i=1:n, j=1:m)
  a(i,j) = 0.0
  b(i,j) = 1.0
  c(i,j) = 1.0 / (i+j+1)  !
  d(i,j) = SQRT(c(i,j))   ! dependence
END FORALL
```

À l'intérieur de ces boucles, des fonctions sans effets de bords peuvent être appelées, comme des fonctions intrinsèques de calcul par exemple. L'utilisateur peut déclarer lui-même certaines fonctions comme ayant ces propriétés, au moyen de la directive `PURE`, ce qui permet alors de les utiliser dans les boucles parallèles.

La critique que l'on peut faire à la sémantique à parallélisme de données est qu'elle est assez inhabituelle et contre-intuitive pour les utilisateurs habitués au séquentiel, pour lesquels le sens de parallèle s'applique aux boucles dont les itérations sont indépendantes les unes des autres...

La directive INDEPENDENT désigne justement les boucles parallèles parfaitement emboîtées, dont les itérations sont indépendantes les unes des autres, c'est à dire qu'elles peuvent être exécutées dans n'importe quel ordre, y compris de façon entremêlée.

Par exemple, si le programmeur dispose d'informations sur une indirection :

```
CHPF$ INDEPENDENT
      DO i=1,100
        A(P(i)) = B(i)  ! P permutation
      END DO

CHPF$ INDEPENDENT
      FORALL (i=1:n)
        A(P(i)) = B(i)  ! idem
      END DO
```

Les variables locales au corps de boucle peuvent également être précisées dans la directive :

```
CHPF$ INDEPENDENT(i,j), NEW(TEMP)
      DO 10 i=1,n
        DO 10 j=1,n
          TEMP = 0.0
          DO 20 k=1,n
 20         TEMP = TEMP + A(i,k)*B(k,j)
          C(i,j) = TEMP
 10   CONTINUE
```

Les boucles parallèles de HPF proposent donc une grande variété de sémantiques, qui satisfont tant les constructeurs de machines à architecture SIMD (sémantique à parallélisme de données du FORALL), que les constructeurs de machines MIMD (sémantique parallèle usuelle de la directive INDEPENDENT).

Cependant le principe de compatibilité **descendante** des codes en HPF (un programme en HPF peut être compilé par un compilateur Fortran 90) n'a pas été respecté, puisque de nouvelles instructions ont été ajoutées. De plus, le problème de la localisation du calcul sur les processus n'a pas été précisé (*i.e.* pas de spécification ON pour une boucle parallèle, comme Fortran D ou Vienna Fortran).

3.2.5 Entrées et Sorties

malgré d'importantes discussions, aucune décision n'a été prise concernant les Entrées-Sorties, et le document rédigé par le sous groupe chargé de cette réflexion n'a pas été retenu par le Forum. Il ne fait pas partie de la spécification HPF. Il est apparu au cours des débats que ces problèmes dépendent pour l'instant beaucoup de l'architecture matérielle de la machine parallèle, et qu'aucun des modèles proposés ne permettait de s'abstraire suffisamment de ces aspects.

3.2.6 Fonctions intrinsèques

Ces nouvelles fonctions ajoutées par le Forum à celles de Fortran 77 et Fortran 90 portent sur :

- le système, en autorisant à s'interroger sur le nombre de processeurs disponibles, et leur arrangement

(`NUMBER_OF_PROCESSORS()`, `PROCESSORS_SHAPE()`);

- le placement des données
  (`HPF_ALIGNMENT()`, `HPF_TEMPLATE()`, `HPF_DISTRIBUTION()`);
- ou des extensions de fonctions intrinsèques déjà existantes (`MINLOC()`, `MAXLOC()`, etc.);
- une librairie nouvelle de fonctions est prévue, baptisée `HPF_LIBRARY`, qui ajoute de nouvelles fonctions de base et propose tout un jeu d'extensions aux fonctions de réduction du langage.

3.2.7 Procédures extrinsèques

Des possibilités d'interfaçage de HPF avec des fonctions écrites dans d'autres langages, ou d'autres style de programmation, sont prévues, au moyen de déclaration spécifiques des procédures appelées. L'idée est de pouvoir optimiser certaines parties critiques de programme à la main, par échange de message par exemple, pour améliorer les performances.

3.2.8 Sous-ensemble officiel de HPF

Afin de faciliter l'apparition des premiers compilateurs, le Forum, conscient des difficultés de ces développements, a décrit un sous-ensemble officiel du langage, le *HPF Subset*. Ce sous-ensemble du langage est :

- basé sur **Fortran 77** ;
- retient de **Fortran 90** :
  - quelques constructions et instructions (`DO WHILE`, etc.);
  - des fonctions intrinsèques ;
  - les fonctions de réduction (`MAXVAL()`, `SUM()`, etc.);
  - les blocs d'interface ;
  - la notation des sections de tableau ;
  - le `WHERE` (sémantique à parallélisme de données);
  - l'allocation dynamique (`ALLOCATE`).
- et comprends les extensions de **HPF sauf**:
  - les directives dynamiques ;
  - la construction `FORALL` ;
  - la `HPF_LIBRARY` et les nouvelles fonctions intrinsèques.

Ce sous ensemble appelle quelques commentaires : d'abord l'exclusion des directives dynamiques explicites n'enlève pas le concept au niveau du compilateur, puisque le travail de réarrangement des données devra être implémenté au niveau des appels de sous-routines qui contiennent comme paramètres des tableaux répartis entre les processeurs. Ensuite, le langage à compiler reste malgré tout important. Les problèmes d'optimisation sur machines parallèles sont difficiles, et la multiplicité des structures impliquées ne facilitera pas cette phase.

3.2.9 Conclusion

Les différents aspects de HPF n'ont pas la même importance. Il est bien évident que le point essentiel de la proposition sera le placement des données. Cette aspect de HPF conditionne sa réussite, qui doit être suffisamment large pour permettre de spécifier des placements de données différents, susceptibles de s'adapter à de nombreux problèmes. Cependant, si son pouvoir d'expression est trop grand, les optimisations seront plus difficiles, les codes produits moins efficaces, les utilisateurs moins contents. De plus, une des conditions supplémentaires du succès de HPF est que les extensions soient suffisamment accessibles et compréhensibles par les utilisateurs. Ces multiples conditions plutôt contradictoires ont rendu difficiles la spécification de la partie placement des données du *High Performance Fortran*. Enfin, on peut s'interroger sur les chances de réussites d'un langage conçu en comité et décidé à force de votes majoritaires et de nécessités commerciales.

3.3 Directives HPF pour le Placement de données

Voici une présentation des directives HPF pour le placement de données, d'après la proposition de Guy L. Steele Jr. Syntaxiquement, celles-ci ont la forme de commentaires, ce qui permet de compiler des programmes HPF avec un compilateur normal sans adaptation, ce qui peut être très utile, en phase de déboguage par exemple. On ne retiendra (sous section 3.3.2) qu'un sous ensemble syntaxique des directives statiques, afin de simplifier la partie analyse syntaxique (*parsing*) de l'implémentation.

3.3.1 Proposition de Guy L. Steele Jr.

La programmation des ordinateurs parallèles à mémoire répartie, et plus généralement des machines parallèles à mémoire non uniforme, s'est avérée très difficile, car elle nécessite la programmation au moyen de messages échangés entre les processus. L'idée reprise par le HPF-Forum est de permettre aux programmeurs de spécifier de quelle manière doivent être placées les données entre les processeurs, et de laisser la génération des messages et des synchronisations aux compilateurs. Le sous-groupe *distribute*, dont Guy L. Steele Jr. organisait les débats, a donc engagé une réflexion pour proposer un ensemble de directives permettant ce placement. Cette correspondance est assurée avec un niveau intermédiaire appelé *template*. La présence ou non de cet intermédiaire fut un sujet de discorde à l'intérieur du sous-groupe.

3.3.1.1 Idée de la proposition

La proposition de Guy L. Steele Jr. s'appuie sur trois objets différents pour décrire la position des éléments d'un tableau sur les processeurs d'une machine parallèle. Ces trois objets sont des tableaux cartésiens (figure 3.4).

Les tableaux de données sont d'abord considérés. Le but est de préciser pour chaque élément un ensemble de processeurs sur lesquels il devra être présent. On considère ensuite des tableaux virtuels, les *templates*, virtuels dans le sens où ils ne correspondent à rien de physique. Les éléments des tableaux sont mis en correspondance avec un ou plusieurs éléments d'un *template*, par un *alignement*. Enfin le troisième objet à considérer est un arrangement de *processors*. On désigne ainsi une vue cartésienne des processeurs de la machine. Chaque élément de *template* est mis en correspondance avec un élément de l'arrangement de *processors*, en spécifiant une *distribution*.

Cette approche permet notamment de demander au compilateur de répliquer certaines données sur plusieurs processeurs. Elle permet de suivre à la trace de manière naturelle les éléments, et donc de déterminer facilement l'ensemble des processeurs qui ont une copie locale de la valeur de cet élément. Par contre la structure mathématique n'en est pas très simple.

3.3.1.2 Alignement

Un **template** est un tableau cartésien vide, un tableau de rien. L'idée de l'alignement de données sur un *template* est que les données alignées sur un même élément de *template* devront finalement se retrouver sur un même processeur. La directive `ALIGN` permet de spécifier un tel alignement pour un tableau ou un scalaire. La directive `REALIGN` permet de modifier cette spécification de manière dynamique.

Un alignement a lieu entre les dimensions d'un tableau et celles d'un *template*. La fonction d'alignement est avant tout une fonction affine d'une variable muette décrivant les indices possibles d'une dimension du tableau aligné, et à valeur dans les indices de la dimension du *template* concerné. La proposition n'autorise pas, en cas de débordement, de boucler sur le début de la dimension du *template*. Cette possibilité avait été dans un premier temps comsidérée, mais aurait inutilement complexifié le traitement, si tel était le cas.
Exemple :

```
      REAL A(10)
!HPF$ TEMPLATE T(30)
!HPF$ ALIGN A(I) WITH T(2*I+5)
```

Dans cet exemple, les éléments du tableau `A` sont mis en correspondance avec des éléments du *template* `T`, avec une fonction affine.

Le *template* est un point de rendez-vous entre plusieurs tableaux. Dans les autres propositions utilisant les alignements, mais pas les *templates*, c'est un des tableaux qui sert ainsi de ciment. L'avantage du *template* est qu'il peut être plus grand que tous les tableaux utilisés dans le programme. Par contre, il présente l'inconvénient d'ajouter un concept de plus au langage.
Exemple :

```
C deux vecteurs pour les bords et une matrice pour le centre...
      REAL A(10), B(10), C(8,10)
!HPF$ TEMPLATE T(10,10)
!HPF$ ALIGN A(J) WITH T(1,J)
!HPF$ ALIGN C(I,J) WITH T(I+1,J)
!HPF$ ALIGN B(J) WITH T(10,J)
```

Le *template* `T` permet de recoller ensemble les vecteurs `A` et `B`, et la matrice `C`. De plus, on peut ne pas aligner certaines dimensions d'un tableau, et ne pas préciser d'alignement pour certaines dimensions du *template*, an moyen de la notation `*` :

```
      REAL A(100,100)
CHPF$ TEMPLATE T(100,100)
CHPF$ ALIGN A(I,*) WITH T(I,*)
```

qu'il faut interpréter ainsi : $\forall i, j, k$ valides, $a(i, j)$ est aligné avec $t(i, k)$.

Il y a donc duplication des valeurs du tableau `A` sur la seconde dimension du *template* `T`, et le placement est le même pour toutes les lignes de `A`.

3.3.1.3 Distribution

Les éléments de *template* sont distribués sur les *processors*, qui sont également arrangés en tableau. La directive `DISTRIBUTE` assure la déclaration d'une distribution de manière statique et `REDISTRIBUTE` le fait dynamiquement.

La distribution d'un *template* se fait par dimension entre le *template* et l'arrangement de *processors*. Pour une dimension donnée du *template* distribué, on spécifie la taille des paquets d'indices contigus de cette dimension du *template* qui doivent être mis en correspondance avec un même indice sur la dimension de l'arrangement de *processors* concerné, au moyen des mots `BLOCK` et `CYCLIC`. Dans le cas de `CYCLIC`, on boucle autour de la dimension de l'arrangement de *processors*, si nécessaire et sans restriction, alors que ce n'est pas possible dans le cas de `BLOCK`.
Exemple :

```
      REAL A(10)
!HPF$ TEMPLATE T(10)
!HPF$ PROCESSORS P(2)
!HPF$ ALIGN A(I) WITH T(I)
!HPF$ DISTRIBUTE T(CYCLIC(3)) ONTO P
```

Dans cet exemple, le processeur `P(1)` obtient les éléments du tableau `A` de 1 à 3 et de 7 à 9, tandis que `P(2)` récupère les éléments 4 à 6 et le 10.

La philosophie des `BLOCK` et `CYCLIC` est la suivante : la distribution permet au programmeur de préciser le grain du parallélisme qu'il souhaite exploiter sur les processeurs de la machine cible. Le choix optimal dépend de la machine et de l'algorithme utilisé. Les différentes distributions permettent de chercher à minimiser le surcoût de contrôle d'une part, et d'assurer le partage de charge d'autre part.

`BLOCK` : permet de répartir les données au maximum entre les processeurs, de manière à minimiser les communications engendrées par des algorithmes qui accèdent au voisinage des points à chaque itération.

`BLOCK(n)` : permet d'ajuster la taille des blocs de façon à trouver le meilleur ratio possible entre communications et calculs, mais sans que plusieurs blocks puissent être affectés à un même processeur.

`CYCLIC` : homogénéise la répartition de charge entre les processeurs dans le cas d'algorithmes accédant à des parties seulement des tableaux (pour des matrices triangulaires par exemple, `BLOCK` donnerait une charge importante à certains processeurs).

`CYCLIC(n)` : permet d'ajuster la taille pour les algorithmes ayant à la fois des accès au voisinage et non homogènes, c'est à dire d'atteindre aussi un objectif de répartition de charge.

3.3.1.4 Remarques

Les directives permettent de spécifier la duplication d'éléments de tableau le long de dimensions de *templates*. Cette possibilité est intéressante car elle permet de demander la duplication d'éléments souvent utilisés par les processus, et gagner ainsi du temps en évitant des communications. Le modèle de machine choisit par le Forum a conduit à identifier pratiquement les concepts de processus et de processeur. Le sous-groupe *distribute* a également discuté des problèmes de passage des paramètres aux fonctions ou sous-programmes lorsque les variables sont réparties.

Un débat important a eu lieu à l'intérieur du sous-groupe, entre partisans et adversaires des *templates*. Il est intéressant de donner un échantillon des arguments développés de part et d'autre.

Pour les *templates* :

- ils permettent d'assurer un lien entre les tableaux qui doivent être répartis ensemble, et peuvent ainsi l'être en une seule directive ;
- un *template* peut être plus grand qu'aucun des tableaux utilisés dans le programme, ce qui est commode dans certains cas ;
- le portage d'applications est plus facile ainsi : le concept d'alignement dépend de la structure du problème, et a peu de chance de devoir être modifié en cas de changement de machine ; par contre, les distributions, qui définissent le grain du parallélisme au niveau des processus, ont de fortes chances de devoir changer lors d'un portage, car le grain optimal dépend de l'architecture ;
- les *templates* sont jolis, confortables, intuitifs, bref mieux pour les utilisateurs.

Contre les *templates* :

- le pouvoir d'expression avec et sans *templates* est le même ;
- c'est un niveau de plus entre les données et les processeurs, ce qui complique le modèle sous-jacent de mise en correspondance des données avec les processeurs ;
- lors des développement ultérieurs de HPF, les *templates* seront un poids difficile à gérer convenablement ;
- l'absence d'arguments « convainquants » pour les *templates* est un argument convainquant contre les *templates* !
- des problèmes se posent aux interfaces : la décision d'assurer la compatibilité du code avec des compilateurs (standards) qui ne prennent pas en compte les directives implique que les *templates* ne peuvent être passés entre procédures, d'où quelques difficultés ;
- les *templates* ne peuvent pas être passés en paramètres lors des appels de fonctions, dans la proposition de Guy Steele.

Les *templates* ont néanmoins été retenus dans la proposition finale.

3.3.2 Sous-Ensemble Syntaxique des Directives Statiques

Voici le sous-ensemble syntaxique des directives statiques concernant le placement des données que l'on a choisi de traiter dans l'implémentation du prototype de compilateur `hpfc`. On retrouve toute la sémantique de la proposition de Guy L. Steele Jr. Le sucre syntaxique a simplement été retiré. On précise également quelques contraintes.

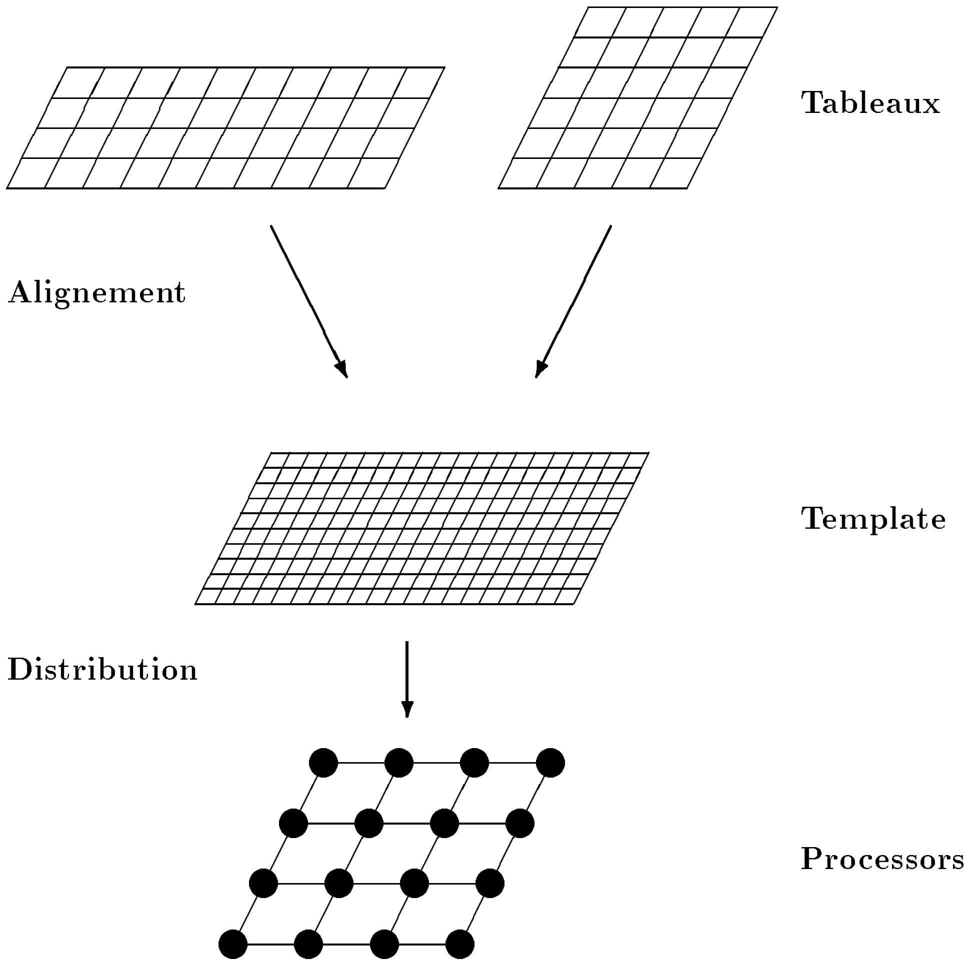

FIG. 3.4 - *Visualisation du placement de données*

3.3.2.1 Syntaxe simplifiée

Les deux premières directives permettent de déclarer des arrangements de *processors* et de *templates*. Les deux suivantes spécifient les alignements et les distributions des variables, le plus souvent des tableaux. I représente ici n'importe quelle variable muette. *int* désigne une constante entière.

```
CHPF$ PROCESSORS P(int, ...)[, ...]
CHPF$ TEMPLATE T(int, ...)[, ...]
CHPF$ ALIGN A(I or *,...) WITH T(int*I+int or *,...)
CHPF$ DISTRIBUTE T(BLOCK(n) or CYCLIC(n) or *,...) ONTO P
```

On exclut donc pour l'instant des déclaration telles que :

```
      parameter(N=100)
      real, dimension(N,N) :: A, B, C
chpf$ template, dimension(N, N), distribute(block, block) :: T
chpf$ align (I,J) with T(I,J) :: A, B, C
```

qui peuvent être réécrites sous la forme :

```
      parameter(N=100)
      real A(N,N), B(N,N), C(N,N)
chpf$ template T(N,N)
chpf$ distribute T((block, block)
chpf$ align A(I,J), B(I,J), C(I,J) with T(I,J)
```

3.3.2.2 Contraintes

Tout d'abord l'utilisation des variables muettes et des constantes doit vérifier certaines conditions :

- les variables muettes ne doivent apparaître qu'une seule fois de chaque côté ;
- les *int* doivent être des entiers explicités

  (pas de `PARAMETER` gérés hors des déclarations, ni d'appels à des fonctions intrinsèques).

Afin d'éviter d'avoir à traiter des cas limites, on suppose de plus :

- qu'un *template* doit avoir au moins une dimension, et qu'il est non vide ; (RD_1)
- que la même condition doit être respectée par les arrangements de *processors*. (RD_2)

3.4 Formalisation des directives HPF

Les alignements (`ALIGN`) et distributions (`DISTRIBUTE`) spécifiables dans HPF ont une structure mathématique simple qu'il est possible de décrire précisément. Cette structure s'appuie essentiellement sur l'algèbre linéaire, ainsi que quelques opérations plus ou moins difficiles à optimiser comme la division entière ou le modulo. Le but de cette section est de formaliser et de proposer une formulation mathématique de la méthode de placement de données proposée par Guy Steele dans le cadre du sous-groupe *distribute* du HPF-Forum, qu'il a animé tout au long de l'année 92. Une démarche de formalisation mathématique similaire a été entamée par O'Boyle et Hedayat [OH92] pour discuter du concept d'alignement et guider l'améliorer du placement pour assurer la localité des données. Cette formalisation a été adaptée et adoptée dans [ACIK93].

3.4.1 Définitions

Voici tout d'abord quelques définitions utiles qui permettent de donner une description propre et claire des différents niveaux pour la distribution et le placement des données en HPF. On définit un espace d'indice, une matrice de sélection et des classes d'objets.

3.4.1.1 Espaces d'indices de dimension n

Un espace d'indices représente les valeurs que peuvent prendre les indices d'accès aux éléments d'un tableau cartésien. Dans la proposition de Guy Steele, tous les objets considérés sont des tableaux cartésiens (*arrays, templates, processors*). On a donc formellement : un ensemble d'indices de dimension n ($n \in N$), noté $[\bar{\imath}_m, \bar{\imath}_M]$, avec $(\bar{\imath}_m, \bar{\imath}_M) \in (Z^n)^2$:

$$[\bar{\imath}_m, \bar{\imath}_M] \;=\; \{\bar{\imath} \in Z^n \mid (\forall j \in [1 \ldots n])\,(\bar{\imath}_m(j) \leq \bar{\imath}(j) \leq \bar{\imath}_M(j))\}$$

L'ensemble des espaces d'indices de dimension n est noté $\mathcal{I}_n$. Un espace est non vide si il existe un vecteur d'entiers relatifs qui satisfait les contraintes. L'ensemble des espaces d'indices non vides est noté de manière naturelle $\mathcal{I}_n^*$.

Exemple : soit $\bar{\imath}$ un vecteur de taille trois dans un espace d'indices $[\bar{\imath}_m, \bar{\imath}_M]$ tel que

$$\bar{\imath}_m \;=\; \begin{pmatrix} 1 \\ -1 \\ 0 \end{pmatrix}, \bar{\imath}_M \;=\; \begin{pmatrix} 11 \\ 10 \\ 5 \end{pmatrix}$$

on a alors

$$\bar{\imath}_1 \;=\; \begin{pmatrix} 2 \\ 9 \\ 3 \end{pmatrix}, \bar{\imath}_2 \;=\; \begin{pmatrix} 5 \\ -2 \\ 5 \end{pmatrix}, \bar{\imath}_1 \in [\bar{\imath}_m, \bar{\imath}_M], \bar{\imath}_2 \notin [\bar{\imath}_m, \bar{\imath}_M]$$

en effet, pour $\bar{\imath}_1$, les valeurs de ses indices sont dans les intervalles autorisés. Par contre, $\bar{\imath}_2$ a le second indice hors limite, et n'appartient donc pas à cet espace d'indices.

3.4.1.2 Matrice de sélection de p vers n

Une matrice de sélection $\bar{\bar{\alpha}}$ de Z^p vers Z^n, est une matrice sur $\{0, 1\}$ telle que la matrice permette de sélectionner n éléments d'un vecteur de taille p, en multipliant cette matrice par ledit vecteur. Il faut donc un élément non nul par lignes.

$$(\bar{\bar{\alpha}} \in Mat_{Select}(n,p)) \;\Leftrightarrow\; (\bar{\bar{\alpha}} \in Mat_{\{0,1\}}(n,p) \mid (\forall i \in [1 \ldots n]\; \exists! j \in [1 \ldots p]\; \bar{\bar{\alpha}}(i,j) = 1))$$

La matrice de sélection est *stricte* si un élément ne peut être sélectionné qu'une fois, ce qui nécessite $n \leq p$. On a alors un élément non nul par colonne au plus. La matrice est dite *normale* si l'ordre relatif des éléments est le même pour les deux vecteurs.

On définit la matrice de sélection *complémentaire stricte* d'une matrice de sélection stricte comme étant la matrice qui sélectionne les colonnes non sélectionnées par la matrice initiale. Il y a unicité si on spécifie que la matrice complémentaire stricte doit également être normale. L'idée intuitive est que la matrice complémentaire permet de sélectionner les dimensions d'un vecteur qui ne sont pas sélectionnés par la matrice de sélection initiale.

Propriété évidente : l'opérateur de complémentarité stricte normale est involutif pour les matrices de sélection strictes normales.

Exemple : on sélectionne du vecteur $\overline{\jmath}$ un sous vecteur $\overline{\imath}$ grâce à la matrice de sélection $\overline{\overline{\alpha}}$.

$$(\overline{\imath} \in Z^n,\ \overline{\jmath} \in Z^p,\ \overline{\overline{\alpha}} \in Mat_{Select}(n,p))\ (\overline{\imath} = \overline{\overline{\alpha}}\,\overline{\jmath})$$

ainsi en considérant $\overline{\jmath}$ un vecteur de taille trois, et $\overline{\imath}$ un vecteur de taille deux, ainsi que les matrices de sélection de 3 vers 2 $\overline{\overline{\alpha}}_1$, $\overline{\overline{\alpha}}_2$ et $\overline{\overline{\alpha}}_3$ suivantes :

$$\overline{\jmath} = \begin{pmatrix} j_1 \\ j_2 \\ j_3 \end{pmatrix}, \overline{\overline{\alpha}}_1 = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, \overline{\overline{\alpha}}_2 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \overline{\overline{\alpha}}_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

on peut alors sélectionner à partir du vecteur $\overline{\jmath}$ des vecteurs de taille deux :

$$\overline{\imath}_1 = \overline{\overline{\alpha}}_1\,\overline{\jmath} = \begin{pmatrix} j_3 \\ j_1 \end{pmatrix}$$

La matrice $\overline{\overline{\alpha}}_1$ sélectionne deux indices sur les trois du vecteur initial. Elle est stricte car il n'y a pas de répétition des indices sélectionnés. Elle n'est par contre pas normale, car le troisième indice de $\overline{\jmath}$ est avant le premier dans le vecteur sélectionné $\overline{\imath}_1$.

$$\overline{\imath}_2 = \overline{\overline{\alpha}}_2\,\overline{\jmath} = \begin{pmatrix} j_2 \\ j_2 \end{pmatrix}$$

Ici, la matrice n'est pas stricte, car la second indice est sélectionné deux fois.

$$\overline{\imath}_3 = \overline{\overline{\alpha}}_3\,\overline{\jmath} = \begin{pmatrix} j_1 \\ j_2 \end{pmatrix}$$

Ce dernier exemple présente une matrice de sélection stricte normale. Il n'y a pas de répétition d'un indice, et l'ordre est respecté.

La matrice $\overline{\overline{\alpha'}}_3 = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$ est sa matrice de sélection complémentaire stricte normale. Elle sélectionne l'indice qui n'était pas sélectionné par la matrice $\overline{\overline{\alpha}}_3$.

3.4.1.3 Classes d'objets

On distingue **trois** classes d'objets, qui sont des espaces d'indices : les *arrays*, les *templates* et les *processors*. Dans la proposition de Guy Steele, les *arrays* sont les tableaux physiques, qui correspondent à des zones de mémoires allouées. Les *templates* sont des *tableaux de rien*, qui n'apportent que la commodité d'un espace d'indices sans impliquer une allocation de mémoire derrière. Enfin les *processors* désignent une représentation cartésienne des processeurs physiques de la machine.

Tout *array* doit avoir un alignement-HPF sur un *template*, et tout *template* doit avoir une distribution-HPF sur un *processors*.

Soit $\mathcal{A}$, les éléments d'un tableau, $\mathcal{T}$ les éléments d'un *template* et $\mathcal{P}$ les éléments d'un *processors*, le premier étant aligné-HPF sur le second, et le second étant distribué-HPF sur le troisième. On a alors les prédicats :

$$a \in \mathcal{A}, t \in \mathcal{T}, p \in \mathcal{P}, Aligned(a,t), Distributed(t,p)$$

Ces prédicats sont vrais quand les deux éléments désignés sont respectivement alignés ou distribués l'un sur l'autre.

3.4.2 Alignement-HPF

On définit la relation d'alignement (alignement-HPF) d'un espace d'indice sur un autre de la manière suivante : Un alignement-HPF d'un espace d'indice (*array*) ($[\overline{a}_m, \overline{a}_M] \in \mathcal{I}_p^*$) sur un autre (*template*) ($[\overline{t}_m, \overline{t}_M] \in \mathcal{I}_n^*$) est un quadruplet $(\overline{\overline{\alpha}}, \overline{\beta}, \overline{\overline{r}}, \overline{\overline{s}})$ tel que :

- $\overline{\overline{\alpha}} \in Mat_Z(q,q)$, une matrice d'entiers relatifs diagonale,
- $\overline{\beta} \in Z^q$, un vecteur entier.
- $\overline{\overline{r}} \in Mat_{Select}(q,n)$, une matrice de sélection stricte normale,
- $\overline{\overline{s}} \in Mat_{Select}(q,p)$, une matrice de sélection stricte,

et qui définit la relation entre ($\overline{a} \in [\overline{a}_m, \overline{a}_M]$) et ($\overline{t} \in [\overline{t}_m, \overline{t}_M]$) :

$$\overline{\overline{r}}(\overline{t} - \overline{t}_m) = (\overline{\overline{\alpha}}\,\overline{\overline{s}}(\overline{a} - \overline{a}_m) + \overline{\beta}) \tag{3.1}$$

Cette relation permet de définir $\overline{\overline{r}}\,\overline{t}$. Les éléments non définis par la relation (*i.e.* $\overline{\overline{r'}}\,\overline{t}$ où $\overline{\overline{r'}}$ est la matrice complémentaire stricte normale de $\overline{\overline{r}}$) ne sont pas spécifiés, et sont contraints uniquement par l'espace d'indices dans lequel est pris $\overline{t}$. On obtient alors un espace d'indice inclus dans $[\overline{t}_m, \overline{t}_M]$ qui représente l'ensemble des éléments de *template* sur lesquels est aligné $\overline{a}$.

$\overline{\overline{r}}$ permet de sélectionner les dimensions de *template* sur lesquelles ont lieu les alignements. Elles sont mises en correspondance avec certaines dimensions du tableau aligné ($[\overline{a}_m, \overline{a}_M]$) au moyen de la matrice de sélection $\overline{\overline{s}}$. $\overline{\overline{\alpha}}$ et $\overline{\beta}$ précisent la relation affine d'alignement

L'intérêt de cette description est qu'elle permet par un jeu de création et d'écrasement de dimensions de répliquer ou de regrouper les données sur les éléments de *template*. Les éléments des *arrays* sont mis en correspondance avec des éléments de *template*.Voici quelques exemples :

3.4.2.1 Transposition

```
      REAL A(10,10)
CHPF$ TEMPLATE T(20,20)
CHPF$ ALIGN A(I,J) WITH T(J,I)
```

On a alors l'alignement :

$$(\overline{\overline{\alpha}}, \overline{\beta}, \overline{\overline{r}}, \overline{\overline{s}}) = \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right)$$

On remarque que la dernière matrice n'est pas diagonale, ce qui est dû à l'inversion des variables muettes `I` et `J` entre le tableau `A` et le *template* `T` dans la directive d'alignement (`ALIGN`).

3.4.2.2 Relations Affines

```
      REAL A(10,10)
CHPF$ TEMPLATE T(20,10)
CHPF$ ALIGN A(I,J) WITH T(2*I-1,J+7)
```

On a dans ce cas l'alignement :

$$(\overline{\overline{\alpha}}, \overline{\beta}, \overline{\overline{r}}, \overline{\overline{s}}) = \left(\begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} -1 \\ 7 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right)$$

On reconnait sans peine dans la première matrice et le vecteur qui la suit les coefficients des deux fonctions affines qui définissent l'alignement, les matrices de sélection étant simplement l'identité. Supposons que l'on cherche les éléments de *template* sur lesquels se trouve un élément du tableau `A`. En appliquant la formule proposée, on trouve pour un élément donné du tableau `A`, représenté par le vecteur de la valeur des indices, certains des indices du *template* `T` correspondant.

Ainsi en prenant le vecteur d'indice $\overline{a}$, et en considérant les vecteurs qui limitent les espaces d'indices $\overline{a}_m$, $\overline{a}_M$, $\overline{t}_m$ et $\overline{t}_M$, on trouve le vecteur d'indices $\overline{t}$ en appliquant simplement la formule refeqn:align :

$$\overline{a}_m = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \overline{a}_M = \begin{pmatrix} 10 \\ 10 \end{pmatrix}, \overline{t}_m = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \overline{t}_M = \begin{pmatrix} 20 \\ 10 \end{pmatrix}$$

$$\overline{a} = \begin{pmatrix} 4 \\ 5 \end{pmatrix} \Rightarrow \overline{t} = \begin{pmatrix} 7 \\ 2 \end{pmatrix}$$

3.4.2.3 Duplication

```
      REAL A(10)
CHPF$ TEMPLATE T(10,3)
CHPF$ ALIGN A(I) WITH T(I,*)
```

On a dans ce cas l'alignement :

$$(\overline{\overline{\alpha}}, \overline{\beta}, \overline{\overline{r}}, \overline{\overline{s}}) = \left(\begin{pmatrix} 1 \end{pmatrix}, \begin{pmatrix} 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 \end{pmatrix} \right) \quad \overline{\overline{r'}} = \begin{pmatrix} 0 & 1 \end{pmatrix}$$

Le second indice de l'élément de *template* n'est pas spécifié par l'alignement. On considère donc la matrice complémentaire stricte normale de la matrice de sélection qui doit s'appliquer au vecteur d'indice de *template*. On doit alors préciser sur la dimension concernée l'ensemble des indices possibles. On obtient donc à partir d'un élément du tableau `A` de vecteur d'indices $\overline{a}$, les éléments de *template* correspondant :

$$\overline{a}_m = \begin{pmatrix} 1 \end{pmatrix}, \overline{a}_M = \begin{pmatrix} 10 \end{pmatrix}, \overline{t}_m = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \overline{t}_M = \begin{pmatrix} 10 \\ 3 \end{pmatrix}$$

$$\overline{a} = \begin{pmatrix} i \end{pmatrix} \Rightarrow \overline{t} = \begin{pmatrix} i \\ [1 \ldots 3] \end{pmatrix}$$

3.4.2.4 Écrasement

Il s'agit de montrer simplement que lorsqu'une dimension du tableau est écrasée par l'alignement, la valeur de l'indice sur cette dimension n'intervient pas lors du calcul du vecteur d'indices sur le *template*, cet indice étant filtré par la matrice de sélection du vecteur d'indice du tableau ($\overline{\overline{s}}$).

```
      REAL A(10,10)
CHPF$ TEMPLATE T(10)
CHPF$ ALIGN A(I,*) WITH T(I)
```

On a dans ce cas l'alignement :

$$(\overline{\overline{\alpha}}, \overline{\beta}, \overline{\overline{r}}, \overline{\overline{s}}) = \left(\begin{pmatrix} 1 \end{pmatrix}, \begin{pmatrix} 0 \end{pmatrix}, \begin{pmatrix} 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \end{pmatrix} \right)$$

Les bornes des espaces d'indices des tableau et *template* :

$$\overline{a}_m = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \overline{a}_M = \begin{pmatrix} 10 \\ 10 \end{pmatrix}, \overline{t}_m = \begin{pmatrix} 1 \end{pmatrix}, \overline{t}_M = \begin{pmatrix} 10 \end{pmatrix}$$

Ce qui donne :

$$\overline{a} = \begin{pmatrix} i \\ j \end{pmatrix} \Rightarrow \overline{t} = \begin{pmatrix} i \end{pmatrix}$$

3.4.3 Distribution-HPF

L'idée de la distribution-HPF est de regrouper les éléments des dimensions d'un *template* distribués par paquets, et de mettre en correspondance ces dimensions régulièrement découpées avec les dimensions des *processors*. Tout élément de *template* est présent sur un élément de l'espace d'indices des *processors*. On définit la distribution-HPF d'un espace d'indices *template* ($[\overline{t}_m, \overline{t}_M] \in \mathcal{I}_n^*$) sur un autre espace d'indices *processors* ($[\overline{p}_m, \overline{p}_M] \in \mathcal{I}_r^*$) par une paire ($\overline{\overline{q}}, \overline{k}$) telle que

- $\overline{\overline{q}} \in Mat_{Select}(r, n)$ une matrice de sélection stricte normale,
- $\overline{k} \in (N^*)^r$ un vecteur de paramètres de distribution.

et qui permet de lier ($\overline{t} \in [\overline{t}_m, \overline{t}_M]$) avec ($\overline{p} \in [\overline{p}_m, \overline{p}_M]$) de la manière suivante le modulo est la division entière étant à considérer termes à termes :

$$(\overline{p} - \overline{p}_m) = (\overline{\overline{q}}(\overline{t} - \overline{t}_m) \div \overline{k}) \bmod (\overline{p}_M - \overline{p}_m + \overline{1})$$

Par abus, on peut distribuer un espace d'indices en considérant l'image par la distribution-HPF des indices. Il s'agit d'un autre espace d'indices inclus dans celui des *processors*, d'après le modulo proposé dans la formule ci-dessus. Comme l'alignement d'un tableau sur un *template* conduit à attribuer à chaque élément de tableau un sous-espace d'indice du *template*, cette propriété permet de poursuivre directement les opérations, et de décrire simplement le sous-espace d'indices des processeurs qui ont une copie de la valeur de l'élément.

Étant donné trois espaces d'indices, un *array*, un *template* et un *processors*, plus un alignement-HPF et une distribution-HPF liant ces espaces, il est possible de connaître en suivant les descriptions proposées, pour un vecteur donné de l'*array*, l'espace d'indices correspondant sur le dernier (les *processors*). Il correspond à l'ensemble des *processors* sur lesquels est présent l'élément désigné par le vecteur initial.

Exemple :

```
CHPF$ TEMPLATE T(100,100,100)
CHPF$ PROCESSORS P(20,20)
CHPF$ DISTRIBUTE T(CYCLIC(1),CYCLIC(5),*) ONTO P
```

On a alors la Distribution-HPF suivante :

$$(\overline{\overline{q}},\overline{k}) = \left(\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 5 \end{pmatrix}\right)$$

La première matrice sélectionne les deux premières dimensions du *template* `T`, qui sont mise en correspondance avec les deux dimensions de l'arrangement de processeurs `P`. On retrouve simplement dans $\overline{k}$ les paramètres des `CYCLIC` de la distribution.

Les bornes des espaces d'indices des *template* et *processors*:

$$\overline{t}_m = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \overline{t}_M = \begin{pmatrix} 100 \\ 100 \\ 100 \end{pmatrix}, \overline{p}_m = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \overline{p}_M = \begin{pmatrix} 20 \\ 20 \end{pmatrix}$$

Sur la première dimension du *template* `T`, les éléments sont distribués de manière cyclique et par paquet de un sur la première dimension de l'arrangement de *processors* `P`. Les processeurs de la première colonne (`P(1,*)`) se retrouvent donc avec des éléments de *template* dont la valeur de l'indice sur la première dimension est 1, 21, 41, 61 ou 81. Par un même raisonnement, on constate que les processeurs de la première ligne de l'arrangement de *processors* se voit affecté les éléments de *template* dont les indices sur la seconde dimension vont de 1 à 5. La troisième dimension est elle écrasée, et l'effet est comparable à l'écrasement d'une dimension d'un tableau lors d'un alignement sur un *template*. Les éléments de *templates* `T(`i`,`j`,`k`)` sont présents sur les *processors* `P(`i`,`j`)`.

3.4.4 Discussion

L'approche proposée par Guy Steele (l'utilisation d'un niveau intermédiaire, le *template*, la distinction entre alignement et distribution, ...) est intéressante car elle permet à l'utilisateur une grande variété de description d'alignements et de distributions entre tableaux et processeurs. On peut notamment spécifier une certaine dose de duplication des données, ce qui est une nécessité pour exploiter correctement des tableaux ou scalaires dont les valeurs sont souvent sollicités et rarement définies, comme ce peut être le cas pour les conditions aux limites d'une équation qui doit être résolue de manière itérative, et qui sont souvent utilisées dans l'algorithme, alors qu'elles sont fixées au début des itérations.

Cependant on peut se demander dans quelle mesure l'approche adoptée est bonne : en effet, plutôt que de partir exclusivement du souhait des utilisateurs et de se laisser guider par un soucis de simplicité, on aurait pu dans un premier temps s'interroger sur les propriétés que les fonctions de distributions devraient avoir pour permettre une bonne exploitation par le compilateur. Par exemple, est-il souhaitable que ces fonctions soient inversibles? Cette condition peut être fort utile pour espérer faire quelques optimisations lors d'une compilation pour machine à mémoire répartie, en factorisant de manière plus facile les messages générés par le processus de compilation.

Une meilleure approche, sans doute, aurait été de constituer une liste de propriétés souhaitées par les utilisateurs, souhaitables pour les optimisations, et de chercher ensuite des classes de fonctions possédant ces propriétés, ou au moins un certain nombre d'entre elles. Enfin de chercher une manière syntaxique et sympathique d'exprimer ces fonctions.

Il est également important de pouvoir distinguer le possible et l'exploitable. Il faut que les distributions qui provoqueront un surcharge de gestion important soient difficiles à exprimer, et que celles qui seront faciles à gérer et à optimiser se décrivent simplement. Il faut en effet conduire l'utilisateur à programmer de manière à faciliter le travail du compilateur.

La proposition de Guy L. Steele Jr. permet de spécifier des choses fort bizarres. Par exemple on peut préciser les alignement et distribution d'un tableau de manière à mettre deux éléments sur un processeur, puis un sur le suivant, puis à nouveau deux, etc. Ces bizarreries ne sont pas souhaitées, mais le fait est qu'elles sont bien là !
Exemple :

```
      REAL A(10)
CHPF$ TEMPLATE T(20)
CHPF$ PROCESSORS P(7)
CHPF$ ALIGN A(I) WITH T(2*I-1)
CHPF$ DISTRIBUTE T(CYCLIC(3)) ONTO P
```

Dans cet exemple, on met en correspondance par un alignement les éléments du tableau `A` avec les éléments d'indice impair du *template* `T`. Puis la distribution regroupe les éléments de *template* par paquets de trois. En conséquence, chaque bloc contient soit deux éléments d'indice pair, soit deux éléments d'indice impair. D'où la distribution irrégulière du tableau sur l'arrangement de *processors* `P`.

Chapitre 4

Étude d'un schéma de Compilation

4.1 Formalisation

Dans cette section vont être formalisés quelques aspects utiles pour décrire précisément par la suite la compilation. On précise la définition d'un programme, du langage d'entrée du compilateur, du langage de sortie. Le but poursuivi est d'avoir une base permettant de faire quelques démonstrations sur la correction de l'algorithme de compilation, tout en pouvant intégrer de manière continue les optimisations ultérieures et en gardant la possibilité de démontrer que les opérations faites sont sémantiquement correctes, *i.e.* que l'on n'a pas modifié la fonctionnalité du programme initial. La formalisation n'est cependant pas stricte et exhaustive, car il serait un peu long de devoir reprendre toute la sémantique dénotationnelle dans le cadre de ces travaux.

4.1.1 Compilation

L'algorithme de compilation $\mathcal{T}_0$ s'appuit sur un certain nombre d'hypothèses, et poursuit quelques objectifs qui sont présentés ici.

4.1.1.1 Parallélisme

Le langage à compiler est supposé fournir deux types de renseignements. D'abord, des indications sur la manière dont on veut que le parallélisme soit géré, comme le nombre de processus à utiliser, ainsi que la façon dont doivent être réparties les données entre les processus. Ensuite, il doit indiquer où se trouve le parallélisme qui devra être exploité.

La machine cible est une machine MIMD-DM (*Multiple Instructions, Multiple Data, Distributed Memory*). Ce sont les machines les plus difficiles à programmer, mais qui sont également les plus prometteuses au plan ratio puissance/prix. On suppose qu'il existe un processeur distingué appelé *host*, qui est notamment spécialement chargé des Entrées-Sorties, et la machines est formée d'un certain nombre de nœuds que nous appellerons *nodes*.

4.1.1.2 Objectifs de la Compilation

La compilation est vue comme la traduction du programme source de manière à suivre les indications données dans le programme. L'algorithme de base a deux objectifs : il doit assurer la correction sémantique du programme produit, et ce de manière formelle, et servir de base sûre aux versions suivantes optimisées lorsque les optimisations ne seront pas possibles ou pas souhaitables.

Le compilateur produit un programme 2PMD *(2 Programs, Multiple Data)*. Le premier est spécifique au *host*, processeur particulier qui assure les entrées-sorties, et le second est destiné aux nœuds de la machine. Le programme des nœuds est un programme SPMD *(Single Program Multiple Data)* dont l'exécution est paramétrée par l'identité du processeur.

On n' exploite donc pas toutes les possibilités du MIMD. Cependant il est nécessaire de limiter le nombre de sources produits, sans quoi la compilation pour des machines avec un grand nombre de nœuds deviendrait une catastrophe. On peut quand même imaginer limiter à quelques unités le nombre de programmes produits pour améliorer la qualité du code produit.

Fondamentalement, la compilation consistera à parcourir la structure du programme initial, à en filtrer les accès à des variables réparties, et à insérer alors les routines de communications nécessaires.

4.1.2 Programme

Un programme P en entrée du compilateur est défini de la manière suivante : $P = (D, S)$ où D est un ensemble de déclarations ($D \in \mathcal{D}$) et S est un « statement » ($S \in \mathcal{S}$). Cette définition est abstraite et bien loin des directives HPF (`ALIGN`, `DISTRIBUTE`...). On spécifie seulement ce que l'on attend comme informations du programme, indépendamment de la manière dont ces informations sont fournies.

4.1.2.1 Déclarations

Les déclarations d'un programme permettent de définir les variables utilisées, ainsi que des caractéristiques de ces variables quant à l'exécution du programme sous forme de processus distincts, communiquants et synchronisants. Ces déclarations sont composées de déclarations de variables et de directives HPF. La fusion de ces deux concepts est faite pour simplifier l'exposé.

4.1.2.2 Variables

L'ensemble des Déclarations D définit l'ensemble des variables du programme $Var(P)$, que l'on notera simplement Var. Les variables de Var sont des scalaires ou des tableaux de scalaires, basés sur les types int, $real$ et $bool$. Les variables sont distinctes les unes des autres (il n'y a pas d'aliasing). De ce fait les concepts de variable et de case mémoire pour un processus donné sont équivalents.

4.1.2.3 Directives

Les directives amènent un certain nombre de renseignements supplémentaires :

- définition de l'ensemble des processus $\mathcal{P}$ qui exécuteront le programme P ;
- distinction d'un sous ensemble de variables qualifiées de *partagées*, et notées $Part$ ($Part \subset Var$). Un tableau est soit entièrement partagé, soit entièrement privé ;
- les autres variables sont privées, $Priv$. Elles sont locales et internes à chaque processus. Chaque processus accède à la sienne quand sa valeur est nécessaire. Parmi les variables privées on comptera les indices de boucles, les variables temporaires.

  $$(\forall p \in \mathcal{P})\ (Var(p) = Priv(p\ldots) \cup Part) \qquad (A_1)$$

- pour chaque processus p de $\mathcal{P}$, définition de $Part_L(p)$ (l'ensemble des variables Locales au processus) et de $Part_{NL}(p)$ (le complémentaire du précédant *i.e.* les variables Non Locales). Une variable de $Part_{NL}(p)$ n'est pas locale au processus, qui devra donc y accéder d'une manière ou d'une autre, par des communications;

  $(\forall p \in \mathcal{P})\,(Part = Part_L(p) + Part_{NL}(p)) \qquad (A_2)$

  en considérant l'ensemble des éléments de templates T, et les alignements entre les variables partagées et ces éléments d'une part, et d'autre part les distributions entre eux et les éléments de processors, on a :

$$Part_L(p) = \{v \in Part | \exists t \in T, Aligned(v,t), Distributed(t,p)\}$$

On a de plus un certain nombre de propriétés que l'on attend des directives :

- toute variable partagée est locale à au moins un processus;

  $(\forall v \in Part)\,(\exists p \in \mathcal{P} \mid v \in Part_L(p)) \qquad (A_3)$

- de plus on suppose que $\mathcal{P}$ comprend un processus distingué *host*, processus que l'on y ajoute et qui ne ne possède aucune copie de variables partagées. Ce processus servira de base aux entrées/sorties. C'est un moyen de séparer logiquement les parties calculs des autres parties du programme.

  $((\exists host \in \mathcal{P}) \mid (Part_{NL}(host) = Part))\,[\overset{(A_2)}{\Longleftrightarrow}\,((\exists host \in \mathcal{P}) \mid (Part_L(host) = \emptyset))]\;(A_4)$

4.1.2.4 Instruction

Une instruction d'un programme est un élément d'un des ensembles $\mathcal{S}_{in}$, le langage d'entrée, ou $\mathcal{S}_{out}$, le langage de sortie. Ces deux langages s'appuient sur $\mathcal{S}_{core}$ (leur intersection donc). On définit dans les sous-sections suivantes précisément les caractéristiques de ces ensembles.

4.1.3 Core Language $\mathcal{S}_{core}$

Voici le langage qui sert de base aux langages d'entrée et de sortie du compilateur. C'est l'armature d'un langage structuré comparable à Fortran, moins les GOTOs et autres structures qui n'ajoutent rien au pouvoir d'expression du langage, tout en compromettant le travail qui peut être fait automatiquement. Le EXIT présent dans ce langage est une forme moins forte de GOTO qui permet d'imaginer pouvoir gérer des cas particuliers dans un programme. Il est cependant probable que sa présence perturbe les algorithmes de compilation, et qu'aucune bonne compilation ne soit possible simplement lorsqu'il est utilisé. On a également oublié volontairement tous les problèmes de sous-programmes.

4.1.3.1 Expressions $\mathcal{E}$

On dispose d'un sous-langage des expressions, qui n'est pas détaillé ici. On définit la valeur d'une expression comme le résultat d'opérations sur des scalaires. Pour $E \in \mathcal{E}$, ont peut distinguer le type de E : $intE$, $boolE$ ou $realE$. Une opération est une fonction sans effet de bords qui n'utilise que des valeurs de variables (et de constantes) pour opérer, et dont le résultat est une valeur, scalaire. Exemples : $tan()$, $ln()$, mais aussi toute fonction définie par l'utilisateur qui a les propriétés requises. Une sous classe très stricte et réduite de fonctions est donc acceptée, mais il n'est aucunement question d'autoriser des fonctions et procédures arbitraires.

4.1.3.2 Instructions

pour $E \in \mathcal{E}$, $(i, v, def) \in Var^3$, on a $S \in \mathcal{S}_{core}$ si :

S	$::=$	$E \rightarrow def$	Affectation	(c_1)
	$\mid$	S_1 ; S_2	Sequence	(c_2)
	$\mid$	IF $(boolE_1)$ THEN S_1 [ELSE S_2]	Condition	(c_3)
	$\mid$	DO i $(intE_1, intE_2)$ S'	Boucle	(c_4)
	$\mid$	IN() $\rightarrow v$	Entree	(c_5)
	$\mid$	OUT(v)	Sortie	(c_6)
	$\mid$	EXIT	Exception	(c_7)

4.1.3.3 Itération

Afin de distinguer les différentes instructions à l'exécution, on définit l'ensemble des itérations d'une instruction S du programme P noté $It_P(S)$, qui correspond intuitivement à l'ensemble des valeurs que peuvent prendre les indices de boucles englobant l'instruction.

4.1.3.4 Sémantique

Quelques explications :

- def désigne une référence à une variable, qui peut dépendre du résultat de l'évaluation d'expressions (calcul des indices de tableaux...). Cette variable est *définie* par l'affectation.
- exception : cette instruction permet de sortir de la boucle courante.
- le DO est séquentiel. L'index de boucle n'est pas modifié par le contenu de la boucle. Les valeurs des bornes des boucles sont fixées à l'entrée de celles-ci, comme le spécifie la norme Fortran d'ailleurs.
- boucle : $i \in Priv$, (*i.e.* l'indice de boucle fait partie des variables privées du programme). Cette contrainte est posée en ayant à l'esprit le code parallèle à produire. En effet, on souhaite que les différents processus soient le plus indépendants possibles.

4.1.4 Input Language $\mathcal{S}_{in}$

Voici le langage d'entrée du compilateur, $\mathcal{S}_{in}$. Ce langage est basé sur $\mathcal{S}_{core}$.

4.1.4.1 Instruction DOALL

Le langage d'entrée $\mathcal{S}_{in}$ comprend une instruction de plus qui permet de spécifier la présence de parallélisme dans une boucle du programme :

$\mathcal{S}_{core}$ +

$\mid$	DOALL i $(intE_1, intE_2)$ S''	Boucle//	(i_1)

4.1.5 Contraintes

Il y a également un certain nombre de contraintes qu'il faut considérer sur ces instructions. Il ne doit pas y avoir d'indirections de variables partagées dans le programme. C'est à dire de tableau partagé dont l'élément à considérer dépend de la valeur d'une autre variable partagée, comme `A(B(i))`. Les indirections peuvent être réduites par une première phase du compilateur, en insérant des variables privées temporaires qui accueillent les valeurs intermédiaires non locales, comme `T=B(i) ; A(T)`. Ce n'est donc pas a proprement une contrainte sur le langage d'entrée, mais sur le langage soumis à $\mathcal{T}_0$. $\mathcal{T}_0$ supposera donc que le programme a été filtré et que ces réécritures d'indirections ont été faites. Cette phase de normalisation du code est décrite dans la partie traitant de l'implémentation.

S'' peut être une boucle parallèle (Boucle//) avec des bornes indépendantes des variables partagées, ou une instruction de $\mathcal{S}_{core}$, avec cependant quelques restrictions, qui réduisent la généralité de ce qui peut être accepté dans le corps d'un nid de boucles parallèles, et ce dans la but d'en maîtriser facilement la sémantique. L'idée est que le corps d'un nid de boucles parallèles est un ensemble de définitions de variables partagées qui doivent être indépendantes les unes des autres. On peut présenter cela de la façon suivante :

- Il n'y a pas d'effets de bords, pas d'I/O ;
- Le corps de la boucle doit revenir au plan sémantique à l'affectation d'une ou plusieurs variables réparties, qui doivent pouvoir être déterminées à l'entrée dans le corps de boucle ;
- Les variables partagées référencées ou définies doivent directement être connues en fonction de variables privées (il ne doit pas y avoir d'indirections, mêmes implicites). Cette condition est posée pour qu'au début d'une itération le processus puisse connaître les variables partagées qui seront définies ou référencées dans le corps de boucle, afin d'organiser en conséquence les communications.
- Des variables privées peuvent être utilisées à l'intérieur de l'instruction, si la valeur n'est pas utilisée en dehors de l'instruction S'' (*i.e.* soit seule la valeur de la variable privée est utilisée dans le corps de boucle, soit elle peut être redéfinie, mais dans ce cas sa valeur ne pourra pas être réutilisée ultérieurement).
- Il n'y a pas d'exception à une boucle// (*i.e.* pas de EXIT qui sorte du DOALL). (ou alors il s'agit d'un abort et le programme ne finit pas normalement, en d'autres termes une situation catastrophique).
- Remarque : on ne peut donc pas se servir d'une boucle parallèle pour spécifier des opérations parallèles sur des données qui sont des variables privées.

4.1.5.1 Sémantique de la boucle parallèle

Ces contraintes permettent entre autres d'avoir une sémantique de la boucle parallèle (Boucle//) telle qu'elle puisse être remplacée par une Boucle dans un programme sans que l'exécution ne soit changée. Il est intéressant de remarquer que ces contraintes ne sont pas purement syntaxiques, mais que l'on a plutôt cherché à définir un ensemble de propriétés qui doivent être vérifiées.

Les différentes itérations à l'intérieur d'un nid de boucles parallèles sont indépendantes les unes des autres, c'est à dire que le résultat est déterministe, quelque soit l'ordre dans lequel l'exécution est faite. De plus, dans le cas d'un nid de boucles parallèles, l'espace décrit par les indices est fixé à l'entrée de la première boucle parallèle. (Boucle//). Le but de cette restriction est de connaître alors la forme exacte de l'hyperrectangle de points entiers décrit par le nid de boucles parallèles, et de permettre ultérieurement de réduire en fonction de chaque processus la portion d'espace qu'il devra décrire. Si l'on retire cette condition, l'espace à décrire peut alors dépendre de la valeur des l'indices de boucles parallèles englobantes, les formes prises peuvent être fort complexes, et donc plus difficile à exploiter par le compilateur, ce qui n'est pas souhaité pour le niveau prototype visé. Cette restriction ne vise qu'à la simplification du prototype, mais n'apparait pas dans la définition du langage HPF.

L'utilisation des variables partagées et privées est soumise à un certain nombre de contraintes dans une boucle parallèle (Boucle//) : deux itérations ne peuvent pas écrire sur la même variable partagée (exemple, `doall i=1:N t(1)=0` n'est pas correcte). Une variable partagée ne peut être à la fois lue et écrite à des itérations différentes d'une boucle//. Les variables partagées dont on aura besoin des valeurs dans une itération sont connues au début de l'itération. La simple utilisation des valeurs des variables privées n'est pas restreinte. Si une variable privée est lue et écrite, elle est alors locale à l'itération, et sa valeur ne peut être réutilisée ultérieurement.

Le corps d'un nid de boucles parallèles est pris comme un petit programme séquentiel, et devra être exécutée dans un même processus. Cette contrainte suppose qu'il y a des dépendances entre les instructions à l'intérieur d'une série de boucles parallèles (comme par exemple une transposition de matrices...). Pour spécifier les choses autrement, il suffit de mettre une séquence de boucles parallèles. Voir la définition de $Computer()$ (sous section 4.2.2.3).

4.1.6 Output Language $\mathcal{S}_{out}$

Voici le langage cible du compilateur, $\mathcal{S}_{out}$. Il est basé sur $\mathcal{S}_{core}$.

4.1.6.1 Instructions additionnels

$\mathcal{S}_{out}$ comprend un certain nombre de primitives en plus de celles de $\mathcal{S}_{core}$, qui permettent à un processus de s'identifier, de communiquer avec d'autres processus, ainsi que de se synchroniser :

$\mathcal{S}_{core}$ +

|	ME()	Identity	(o_1)
|	SEND(v_{local}) TO p	Send	(o_2)
|	RECEIVE FROM $p' \rightarrow v_{local}$	Receive	(o_3)
|	SYNCHRO (...)	Synchro	(o_4)

4.1.6.2 Contraintes

Quelques restrictions :

- Unicité des identités des processus, *i.e.* il n'y a pas à l'exécution deux processus de même identité (la manière pratique d'assurer cela est une autre paire de manches).

- $p \in (\mathcal{P} \cup \text{ALL})$, le destinataire d'un message peut être un processus de $\mathcal{P}$ ou ALL, qui désigne pour un processus ($\mathcal{P}-$ME()).

- $p' \in \mathcal{P}$, l'origine d'un message est l'un des [autres] processus (cette condition est nécessaire pour la démonstration de l'absence de deadlock dans le programme produit par l'algorithme de compilation).

- v_{local} représente une variable du programme, locale au processus
  ($i.e.\ v_{local} \in (Part_L + Priv)$).

- Il existe un processus d'identité distinguée *host* sur lequel sont faites les I/Os.

- Les I/Os peuvent porter sur des variables privées ou partagées. Vues les restrictions faites sur les variables dont *host* possède une copie, les I/O sur des variables partagées nécessitent des communications.

- Une expression booléenne utilise les opérateurs courants, mais on dispose également du $x \in X$ dans $\mathcal{S}_{out}$ afin de simplifier la description de $\mathcal{T}_0$. Mais on ne dispose pas des ensembles dans le langage. C'est une notation destinée à simplifier la lecture de l'algorithme de compilation.

4.1.6.3 Sémantique

Quelques explications supplémentaires s'imposent :

- ME() désigne l'identité propre d'un processus. ME() $\in \mathcal{P}$. ME() peut intervenir dans des expressions booléennes du programme. $\mathcal{P}$, l'ensemble des processus, est connu de chacun des processus.

- SEND TO envoie la valeur de la variable v_{local} à p. À la fin de l'exécution du Send, la variable a été lue, et elle peut être modifiée sans perturber le message. Le Send est non-bloquant, c'est à dire que l'on est pas sûr que le message soit arrivé à la sortie de l'instruction. Les messages sont supposés bufferisés, il n'y a donc pas de problème à redéfinir une variable qui vient juste d'être envoyée[1].

- RECEIVE FROM reçoit une valeur envoyée par p', et définit v_{local}. À la fin de l'exécution du Receive, la variable a été définie. Le Receive est donc bloquant.

- Si plusieurs messages sont envoyés d'un processus à l'autre, on suppose qu'ils sont reçus dans l'ordre dans lequel ils ont été envoyés. [On ne se préoccupe pas pour l'instant de la méthodologie à employer **pratiquement** pour arriver à un tel résultat. On suppose juste que cela est possible. La question de l'utilisation des numéros de messages de PVM à cette fin est traitée lors de l'exposé sur l'implémentation.]

- les synchronisations ne sont pas utile dans une première phase de prototypage, car le jeu de *send* non bloquants et *receive* bloquants générés dans l'ordre original du programme assure la matérialisation des dépendances entre processeurs, tandis que les dépendances intra-processeur sont conservées par le mode SPMD d'exécution.

1. Les zones tampons de l'implémentation sont celles de PVM 3.

4.2 Compilation Systématique

Voici l'algorithme de compilation $\mathcal{T}_0$ qui insère les communications entre les processus, lorsque celles-ci sont nécessaires.

4.2.1 Esprit de $\mathcal{T}_0$

L'algorithme est une réécriture du programme $P \in \mathcal{S}_{in}$, qui produit le code 2PMD désiré. Les programmes produits sont à exécuter sur un certain nombre de processus qui fonctionnent de manière parallèle, peuvent communiquer et se synchroniser entre eux.

On rappelle que l'on distingue un processus baptisé *host* sur lequel sont faites les I/O. *host* ne possède par ailleurs aucune variable répartie (A_4, section 4.1.2.3). Tous les processus maintiennent valides les variables privées du programme initial ainsi que les copies locales des valeurs des variables *logiquement* réparties. Un seul processus fait les calculs pour une opération qui affecte une variable répartie. En cas de duplication, ce processus envoi la copie de la valeur à tous les processus co-propriétaires. Tous les processus suivent le flot de contrôle. L'algorithme proposé s'adapte normalement bien si l'on considère les directives dynamiques. En effet l'algorithme utilise des descriptions qui sont facilement réactualisable dynamiquement, si les alignements ou distributions sont redéfinies. Il manquerait cependant les transferts nécessaires des valeurs pour faire se correspondre la description et la situation réelle des données réparties.

L'intérêt principal de $\mathcal{T}_0$ est que l'on peut envisager de démontrer formellement sa correction. Cependant des progrès sont à attendre sur le plan performances.

4.2.2 Présentation de $\mathcal{T}_0$

Avant de donner les règles de réécriture de $\mathcal{T}_0$, on en présente le principe, le traitement des déclarations, ainsi que les notations utilisées.

4.2.2.1 Principe

Le statement programme est parcouru en descente récursive (Top-down). Pour chacune des constructions rencontrées, en fonction de conditions connues au moment de la compilation, on compose le code cible, un statement de $\mathcal{S}_{out}$ pour chaque programme, *host* et *node*. Il s'agit d'une simple réécriture récursive de l'arbre syntaxique du programme, avec quelques choix dépendant d'aspects plus sémantiques, comme la nature des variables (privées ou réparties), la maintenance de l'ordre de génération des ensembles qui doit toujours être la même....

4.2.2.2 Déclarations

Le traitement des déclaration consiste à majorer l'espace nécessaire sur chaque processus pour accueillir sa partie des données réparties. Il faut de plus générer les formules d'accès aux indices locaux, ce qui est détaillé en 5.2.4.

4.2.2.3 Owners, Sender, Computer

Ces trois ensembles sont la base des prédicats logiques utilisés dans le processus de réécriture.

Owners Pour une variable scalaire v répartie du programme, $Owners(v)$ désigne l'ensemble des processus qui ont une copie de la valeur de la variable localement : l'ensemble $Owners$ n'est pas défini pour une variable privée. Une erreur est levée si il est utilisé dans ce cadre.

$$(\forall v \in Part)\ (Owners(v) = \{\ p \in \mathcal{P} | v \in Part_L(p)\ \})$$

Sender Pour une variable v répartie du programme, $Sender(v)$ désigne **le** processus qui doit envoyer sa valeur lorsqu'un processus n'en disposant pas localement en a besoin. On s'appuie naturellement sur l'ensemble $Owners(v)$: en effet il est nécessaire que le processus qui décide d'envoyer la valeur en dispose localement.

$$(\forall v \in Part)(\exists Sender(v) \in \mathcal{P} \mid Sender(v) \in Owners(v))$$

Computer *Computer* définit le processus qui doit effectivement faire un calcul. Pour une itération $It(S)$ d'un statement S, les calculs éventuels affectant une variable répartie sont fait par un seul processus. On désigne arbitrairement le $Sender$ de la variable. Dans le cas d'une boucle parallèle, où l'on suppose que les opérations (*i.e.* occurrences de *statements*) d'une itérations sont indépendantes entre elles, on désigne de manière toute aussi arbitraire le $Sender$ de la première variable répartie définie par les opérations.

4.2.2.4 Notations

Voici des précisions sur les notations utilisées pour décrire l'algorithme $\mathcal{T}_0$. Il s'agit d'un jeu de règles de réécriture. Pour chaque constructeur du langage d'entrée $\mathcal{S}_{in}$, on propose la réécriture qui doit en être faite. Les lignes en minuscules marquent des actions que doit faire le compilateur, alors que celles en majuscules représentent le code produit. Lorsque un ensemble est parcouru, le parcourt a toujours lieu dans le même ordre. Pour réduire la taille de la description de l'algorithme, on a abusivement utilisé un SEND TO avec pour destination un ensemble de processus, qui doit se comprendre comme un ensemble de SEND TO du même message à des destinataires différents. Les variables T utilisées par l'algorithme sont des variables privées nouvelles ajoutées aux déclarations du programme. $It(S)$ désigne l'itération courante. Il s'agit de la valeur courante des index de boucle englobant le statement S. Elle doit permettre par exemple dans une boucle parallèle de déterminer précisément les variables réparties qui vont être définies ou lues dans le statement interne au nid de boucles.

L'indentation et l'ajout de ENDIF devrait permettre d'éclaircir le code produit, et rendre les règles de réécriture plus lisibles. La notation $S[T_i/V_i]$ désigne le remplacement dans le statement S des occurences de V_i par des occurences de T_i. Il s'agit d'une substitution totale des variables par d'autres dans le statement.

4.2.3 Règles de réécriture de $\mathcal{T}_0$

Voici les règles de réécriture à appliquer à chaque constructeur, de manière dite Top-down, en partant du statement initial du programme. Voici le résultat de $\mathcal{T}_0\,(S)$ en fonction de chaque constructeur rencontré. Les règles de réécriture sont décrites pour tous les constructeurs du langage $\mathcal{S}_{in}$, tels qu'ils ont été définis dans la formalisation du langage (section 4.1.4).

4.2.3.1 Affectation (c_1, variable répartie)

$S ::= E \rightarrow def$

node :

IF (ME()=$Computer(It(S))$)
THEN
  pour toutes les variables partagees V_i referencees par $(It(S))$
  IF (ME() $\in Owners(V_i)$)
    THEN $V_{i\,local} \rightarrow T_i$
    ELSE RECEIVE FROM $Sender(V_i) \rightarrow T_i$
  fin pour
  $E[T_i/V_i] \rightarrow T_{def}$
  SEND(T_{def}) TO ($Owners(V_{def})$ - ME())
  IF (ME() $\in Owners(V_{def})$) THEN $T_{def} \rightarrow V_{def\,local}$
ELSE
  pour toutes les variables partagees V_i referencees par $(It(S))$
  IF (ME() = $Sender(V_i)$ AND ($Computer$(It(S)) $\notin Owners(V_i)$))
    THEN SEND($V_{i\,local}$) TO $Computer$(It(S))
  fin pour
  IF (ME() $\in Owners(V_{def})$) THEN RECEIVE FROM $Computer(It(S)) \rightarrow V_{def\,local}$
ENDIF

host : $\emptyset$

4.2.3.2 Affectation (c_1, variable privée)

$S ::= E \rightarrow def$

node :

pour toutes les variables partagees V_i referencees par $(It(S))$
IF (ME() $\in Owners(V_i)$)
  THEN
    $V_{i\,local} \rightarrow T_i$
    IF (ME() = $Sender(V_i)$)
      THEN SEND ($V_{i\,local}$) TO (ALL-$Owners(V_i)$)
  ELSE RECEIVE FROM $Sender(V_i) \rightarrow T_i$
  ENDIF
fin pour
$E[T_i/V_i] \rightarrow V_{def}$

host :

pour toutes les variables partagees V_i referencees par $(It(S))$
RECEIVE FROM $Sender(V_i) \rightarrow T_i$
fin pour
$E[T_i/V_i] \rightarrow V_{def}$

4.2.3.3 Séquence (c_2)

$S ::= S_1\, ;\, S_2$

est réécrit pour **host** et **node** en :

$\mathcal{T}_0\,(S_1)\,;$
$\mathcal{T}_0\,(S_2)$

4.2.3.4 Condition (c_3)

$S ::=$ IF $(boolE_1)$ THEN S_1 [ELSE S_2]

est réécrit pour **host** et **node** en :

$\mathcal{T}_0\,(boolE_1 \rightarrow boolT)\,;$
IF $(boolT)$
THEN $\mathcal{T}_0\,(S_1)$
[ELSE $\mathcal{T}_0\,(S_2)$]

4.2.3.5 Boucle (c_4)

$S ::=$ DO i $(intE_1, intE_2)$ S'

est réécrit pour **host** et **node** en :

$\mathcal{T}_0\,(intE_1 \rightarrow intT_1)\,;$
$\mathcal{T}_0\,(intE_2 \rightarrow intT_2)\,;$
DO i $(intT_1$,$intT_2)$
$\mathcal{T}_0\,(S'(i))$

4.2.3.6 Entrée (c_5, variable répartie)

$S ::=$ IN() $\rightarrow v$

host :

IN() $\rightarrow T\,;$
SEND(T) TO $Owners(w)$

node :

IF (ME() $\in Owners(w)$) THEN RECEIVE FROM $host \rightarrow w_{local}$

4.2.3.7 Entrée (c_5, variable privée)

$S ::= \mathrm{IN}() \rightarrow v$

host :

IN() → w ; SEND(w) TO ALL

node :

RECEIVE FROM *host* → w

4.2.3.8 Sortie (c_6, variable répartie)

$S ::= \mathrm{OUT}(v)$

host :

RECEIVE FROM $Sender(w) \rightarrow T$;
OUT(T)

node :

IF (ME() = $Sender(w)$) THEN SEND(w_{local}) TO *host*

4.2.3.9 Sortie (c_6, variable privée)

$S ::= \mathrm{OUT}(v)$

host :

OUT(w)

node : ∅

4.2.3.10 Exception (c_7)

$S ::= \mathrm{EXIT}$

est réécrit pour **host** et **node** en :

EXIT

4.2.3.11 Boucle// (i_1)

$S ::= \mathrm{DOALL}\ i\ (intE_1, intE_2)\ S''$

Pour une boucle//, vues les restrictions posées sur le statement interne, la compilation produit le même genre de code que dans (c_1), quand la variable définie est répartie.

On considère la compilation pour tous les DOALL imbriqués. S'' est donc un statement de $\mathcal{S}_{core}$.

node :

DO i ($intE_1$,$intE_2$)
DO ...

IF (ME() = $Computer(It(S''))$)
THEN
pour toutes les variables partagees V_i referencees par ($It(S'')$)
IF (ME() $\in Owners(V_i)$)
THEN $V_{i\,local} \rightarrow T_i$
ELSE RECEIVE FROM $Sender(V_i) \rightarrow T_i$
fin pour
$S''[T_i/V_i, T_{defi}/V_{defi}]$
pour toutes les variables V_{defi} definies dans $It(S'')$
SEND(T_{defi}) TO ($Owners(V_{defi})$ - ME())
IF (ME() $\in Owners(V_{defi})$) THEN $T_{defi} \rightarrow V_{defi\,local}$
fin pour
ELSE
pour toutes les variables partagees V_i referencees par ($It(S'')$)
IF (ME() = $Sender(V_i)$ AND $Computer(It(S'')) \notin Owners(V_i)$)
THEN SEND($V_{i\,local}$) TO $Computer(It(S''))$
fin pour
pour toutes les variables V_{defi} definies dans $It(S'')$
IF (ME() $\in Owners(V_{defi})$) THEN RECEIVE(V_{defi}) FROM $Computer(It(S'')) \rightarrow V_{defi\,local}$
fin pour
ENDIF
...
ENDDO
ENDDO

host : $\emptyset$

4.2.4 Correction de $\mathcal{T}_0$

Voici quelques éléments informels concernant les démonstrations de correction de $\mathcal{T}_0$. On donne des indications et des idées de théorèmes à démontrer et de la manière de s'y prendre pour ce faire. On trouvera une démonstration complète de correction d'un algorithme de distribution dans le rapport de Bareau, Caillaud, Jard et Thoraval [BCJT92], qui ne s'attache cependant qu'à démontrer la correction de la distribution d'un programme séquentiel, en utilisant le formalisme CSP. L'idée ici est de s'appuyer simplement sur l'algèbre des programmes et la simplicité de la réécriture, pour montrer tout ou partie de la correction de l'algorithme.

4.2.4.1 $\mathcal{T}_0$ termine

L'algorithme $\mathcal{T}_0$ termine sur un statement fini en entrée. Il donne un résultat pour tout programme d'entrée valide.

En effet, soit les règles sont non récursives, soit elles appellent des règles non récursives, soit des règle récursives s'appuyant sur un statement de niveau inférieur (dans l'arbre syntraxique). De plus, la réécriture définie pout tous les cas (chaque constructeur, conditions complémentaires).

4.2.4.2 Petits théorèmes

Quelques théorèmes que l'on aimerait démontrer sous les conditions disponibles :

- les $Owners()$, $Sender()$ et $Computer()$ sont utilisés dans le cadre de leur domaine de définition par $\mathcal{T}_0$.

  vérification point à point (type checking)...

- Vérification des différentes contraintes sur S_{out}.

  vérification pour chaque contrainte et chaque règle...

- toute variable répartie lue localement par un processus lui est locale.

  protection IF (ME() $\in$ *Owners()) THEN ... dans chaque cas*

- Les I/O sont faites sur le processus *host*.

  protection IF (ME()=host) THEN IO dans chaque règle

- Only sender send, sauf IN, pour une variable répartie.

  protection IF (ME() = Sender()) THEN SEND TO dans chaque règle

- hors I/O, only $Part_{NL}(p)$ values are received by a processus p.

  protection $\notin$ *Owners() dans chaque cas, implicitement ou explicitement*

- hors I/O, seules des variables réparties sont échangées

  protection ... dans chaque règle

- aucun message n'est envoyé d'un processus sur lui même.

- équilibre des communications : toute valeur envoyée est reçue par son[ses] destinataire[s].

  symétrie logique des communications générées dans chaque règle, (A_4)...

- il n'y a pas de deadlock dans le programme produit...

4.2.4.3 Justesse des opérations faites

On aimerait montrer que les opérations sont effectuées avec les mêmes valeurs et définissent les même variables que la sémantique de référence.

Idée de la démonstration : par récurrence, en démontrant la conservation d'une précondition liée à la sémantique du programme initial, et qui met en scène les valeurs des variables. Cas particulier de la Boucle// qui doit elle s'appuyer sur la sémantique spécifique spécifiée.

4.3 Analyse des Recouvrements

Cette section présente une version simplifiée de l'analyse des recouvrements pour le prototype de compilateur `hpfc` (*High Performance Fortran Compiler*). L'idée est de faire une analyse des boucles parallèles en liaison avec le placement des données manipulées, et de vérifier si les conditions sont réunies pour générer du code avec utilisation de recouvrements, pour les calculs comme pour les communications. Les déclarations des variables et des bornes de boucles sont à modifier en conséquence.

La méthode utilisée est celle des rectangles [Ger89], par soucis de, simplicité. L'algorithme présenté s'appuie sur de nombreuses hypothèses qui réduisent sa généralité. Le but de cet algorithme est de générer un ensemble de messages, et de déterminer les ensembles de processeurs qui devront échanger ces messages, de manière à assurer les calculs impliqués par une boucle parallèle en suivant la *owner compute rule*. On décrit ici essentiellement l'implémentation de l'algorithme, et il suffit de se rapporter à la littérature citée pour trouver les fondements théoriques.

4.3.1 Conditions

Un certain nombre de conditions sont nécessaires à la validité des transformations faites sur les nid de boucles :

- les nids de boucles parallèles doivent être indépendants, pour être sûr de ne pas faire de fausses transformations en factorisant les messages hors du nid. Les conditions de parallélisme de la boucle impliquent naturellement que le corps de boucle ne contient pas d'entrées-sorties, et que le flot de contrôle soit maintenu à l'intérieur du corps de boucle (pas de `goto` sortant de la boucle par exemple) ;

- accès contigus et rectangulaires des tableaux définis (pour faciliter la génération des parcours des éléments locaux, mais cette condition peut être relachée, si on sait générer le parcours local) ;

- les variables définies (`may` ou `must`) doivent être privées au nid de boucles ou bien réparties (cette condition est de toute façon nécessaire pour que `host` ne soit pas impliqué par les calculs) ;

- les variables réparties qui sont **définies** dans la boucle doivent être alignées entre elles (cette condition permet de s'affranchir d'éventuels problèmes de dépendances internes qui pourraient entrer en conflit avec la *owner compute rule* et la distribution du code) ;

- ces variables ne doivent pas être dupliquées ;

- l'ensemble des variables réparties **définies ou lues** à l'intérieur du corps de boucle doivent être alignées ou presque alignées (*i.e.* les accès doivent être alignés à une translation près) ;

- la distribution doit être par `block`, ou il ne doit pas y en avoir, ou `cyclic` si finalement tout est aligné et qu'il n'y a pas de messages à générer ;

- les alignements doivent se limiter à de simples translations (même si certains cas particuliers pourraient être gérées, comme l'inversion-translation... Il faut limiter la complexité du *align checker*) ;

En plus de ces conditions, inhérentes à l'algorithme, un certain nombre de conditions sont nécessaires pour que l'implémentation fonctionne. On suppose dans un soucis de simplicité que toutes les informations sont disponibles à la compilation, mais beaucoup des informations générées pourraient l'être de la même façon au moment de l'exécution...

- les bornes de boucle du nid de boucle considéré doivent être statiquement connues (ce qui permet de porter au moment de la compilation les calculs de garde par exemple);
- pas de décalage des variables écrites, ce qui implique une normalisation des boucles comme le font Rogers et Pingali [RP89, RP92].

4.3.2 Algorithme

L'algorithme présenté ne vérifie qu'une partie des conditions a priori. Il en vérifie d'autres au fur et à mesure de l'avancement des analyses. Si l'algorithme échoue, parce qu'une condition n'est pas remplie, la méthode de compilation dite de résolution dynamique devra être appliquée à cette partie du code.

4.3.2.1 Notations

On utilisera dans la présentation de l'algorithme de l'analyse des recouvrements qui a été implémenté dans le compilateur `hpfc` les notations suivantes :

- W_d = l'ensemble des références de variables écrites et réparties ; ces références doivent être alignées entre elles, d'après les conditions ;
- R_d = l'ensemble des références de variables lues et réparties.

4.3.2.2 Calcul des vecteurs de recouvrement

À toutes les références de R_d on associe le vecteur de recouvrement qui mesure la distance relative entre les alignements des tableaux écrits (alignés entre eux) et lus (qui doivent être mis en correspondance au moyen de communications). Cette analyse permet de se représenter les données nécessaires à un calcul, relativement au domaine des itérations effectuées localement.

Pour les dimensions distribuées par *block*, soit l'accès est fait à un élément particulier sur la dimension, on qualifie alors la référence sur cette dimension d'absolue (exemple `A(1)`), soit la référence est alignée à un décalage près avec la variable définie, elle est alors qualifiée de relative (exemple `A(i)=B(i+1)`, si les tableaux sont alignés le décalage est de 1).

Pour les dimensions distribuées de manière *cyclic*, on se contente de vérifier qu'il y a alignement, sinon la dimension est marquée comme non alignée, et la boucle ne pourra pas être compilée avec cet algorithme.

Enfin pour les dimensions qui sont locales aux processeurs, on cherche à déterminer quelles sont les indices nécessaires sous la forme d'une section de tableau. Si cette analyse n'aboutit pas et que la référence n'est pas alignée avec les variables définies, le compilateur ne saura pas générer les messages nécessaires de manière statique, et la boucle ne pourra pas là encore être compilée par cet algorithme.

Par exemple dans le cas suivant :

```
      real a(N,N,N,N), b(N,N,N,N)
chpf$ align b with a
```

```
chpf$ distribute a(block,block,*,*)
chpf$ independent(i,j)
      do i,j...
        a(i,2,j,N)=b(i-2,1,j+1,N-1)
      enddo
```

en supposant que sur la seconde dimension les éléments `1` et `2` se retrouvent sur le même processeur, la description du vecteur de recouvrement associé à la référence `b` sera la suivante : `b(`(dist, rel, -2)`,`(dist, abs, 2)`,` (loc, rel, -1)`,` (loc, abs, `N`-1)`)`

4.3.2.3 Analyse des vecteurs de recouvrement

Les vecteurs de recouvrement des dimensions réparties des références de R_d sont analysées de manière à placer ces références dans l'un des ensembles suivants :

- R_a si il y a alignement (pour les termes constants des dimensions réparties, on doit s'assurer statiquement que les données se retrouvent bien sur le même processeur) ;
- R_o (o pour overlap) si les dimensions distribuées le sont par *block* et si il y a presque alignement de ces dimensions, à une translation près.

Le vecteur de recouvrement pour une référence de R_o représente pour les dimensions réparties par *block* la taille et le côté du débordement des accès pour cette référence et les itérations de ce nid de boucle.

Si certaines références ne peuvent être placées dans un de ces ensembles, la boucle ne peut être compilée avec cet algorithme. En toute logique, les messages induits par les références de ces deux ensembles peuvent être vectorisée, et la résolution dynamique appliquée pour obtenir les données manquantes. Cependant les traitements particuliers auxquels ce cas doit conduire n'ont pas été implémentés.

À ce niveau de l'algorithme, on dispose de trois ensembles de références : R_a et W_d qui sont des références alignées entre elles, et R_o qui représente les références qui nécessiteront quelques communications pour assurer la localité des accès.

Les vecteurs de recouvrement des références de R_o sont ensuite parcourus pour vérifier qu'ils sont « acceptables » :

- en contrôlant par exemple qu'un seul voisin sur la dimension est impliqué par l'envoi de messages (par exemple pour des blocks de taille 5 et un recouvrement de taille 8, `a(i)=b(i-8)`, les deux voisins de gauche devraient envoyer des données au processeur...de tels cas ne sont pas traités par l'algorithme) ;
- un autre critère serait de vérifier que les nouvelles déclarations de ces débordements sont compatibles avec l'espace mémoire disponible sur les nœuds de la machine parallèle cible.

Si certaines références de R_o ne sont pas approuvées, l'algorithme échoue.

4.3.2.4 Génération des Messages

Si R_o est vide, il n'y a pas de recouvrement, et on peut passer directement à la génération de code pour chaque *processor*, qui est traitée plus loin. Sinon, il faut générer les messages pour chaque accès, et les insérer dans le code avant de faire les calculs avec la *owner rule*.

Un certain nombre d'étapes sont nécessaires pour générer une description des messages à envoyer effectivement. Il faut définir le contenu des messages individuellement, définir les destinataires relatifs de ces messages (voisins devant recevoir le message) et l'ensemble de *processors* qui devront l'envoyer.

La première étape consiste à générer une première description des messages pour chaque référence de R_o, à partir du vecteur de recouvrement de cette référence :

- à partir de la référence, on génère une description du domaine du tableau référencé qui devra être éventuellement manipulé au moyen de ces messages :

  `do i=3,10 a(i)=b(i-2)` conduit à `b(1:8)` ;

- puis on génère le vecteur de voisinage pour l'arrangement de *processors* à partir du vecteur de recouvrement de la référence. Par exemple :

  ```
  chpf$ align a(i,j,k,l), b(i,j,k,l) with t(j,k,i,l)
  chpf$ distribute t(block, *, block, block) onto p
        do i,j,k,l...
        a (i,j,l,k) = b(i-1, j+2, l, k+3)
  ```

  défini le vecteur de recouvrement : `b(-1, +2, 0, +3)`

  et de voisinage pour `p` : `p(-1, +1, -1)`

- on construit également une description du contenu du message qui devra être envoyé. Cette description est tout à fait virtuelle puisque si le vecteur de voisinage de l'arrangement de *processors* a plus d'un élément non nul, la référence conduira à un ensemble de messages et non pas à un seul message ;

  Avec l'exemple précédent on aurait ainsi, en supposant que la déclaration de `b` :

  `b(1:n1,1:n2,1:n3,1:n4)`

  la description du contenu du message `b(n1:n1,1:2,1:n3,1:3)` ;

L'étape suivante consiste à individualiser les messages, une référence pouvant impliquer différents motifs de communications, si le vecteur de voisinage n'est pas nul sur plus d'une dimension. Exemple, `b(block(10),block(10))` avec `b(i-1,j-1)` implique trois messages : `b(1:9,10)` vers le nord, `b(10,1:9)` vers l'est et `b(10,10)` vers le nord-est.

- pour chaque message « virtuel » précédemment décrit on génère les messages réels qui devront être échangés ; soit d le nombre de dimensions non nulles du vecteur voisinage pour les processeurs de la référence, il y a alors $2^d - 1$ messages à générer ;

- l'algorithme consiste simplement à compter en base 2 pour composer toutes les combinaisons de dimensions retenues ou non sur les d dimensions non nulles du vecteur de voisinage, et à retirer en fin de compte le message pour lequel aucune des dimensions n'était retenue ;

- les messages sont générés à partir de la description initiale faite précédemment : quand une dimension est retenue, la section correspondant sur le tableau reste intacte, et quand elle n'est pas retenue, on prend le complémentaire de cette section pour la dimension considérée.

L'étape suivante consiste, à l'aide des sections décrivant le domaine du tableau concerné, à générer une description de l'ensemble des processeurs qui devront effectivement envoyer ces messages. Pour cela, on suit sur chaque dimension répartie des tableaux les alignement et distribution, en tenant compte du contenu du message (décrit pour les déclarations réduites), ce qui permet d'obtenir un domaine de l'arrangement de *processors* concerné par l'envoi du message :

– le domaine au niveau du tableau est d'abord réduit au minimum nécessaire compte tenu du contenu du message à envoyer.

  Par exemple, si le domaine est `b(3:88)`, pour le tableau `b(1:100)` distribué par `block(10)`, avec le contenu local du message `b(9:10)`, alors le domaine sera réduit à `b(9:80)`, où `9` est le premier éléments du domaine concerné par le message et `80` est le dernier élément du domaine ;

– le domaine peut éventuellement être agrandi par cette phase si le domaine initial comprenait un morceau de message ; l'idée et la conséquence de cette manière de faire est que l'on veut générer un motif d'échange de messages généralisé, sans avoir besoin de traiter les cas particuliers qui peuvent apparaitrent aux bords, et qui nécessiteraient des messages partiellement tronqués ; on se contente alors d'appliquer le motif de communication général, ce qui peut conduire à échanger des données inutiles, voire même non utilisées, mais permet d'éviter la gestion des bords ; il importe juste que les calculs soient limités aux données à définir, et le fait d'avoir envoyer des messages trop grands ne perturbe pas la sémantique du programme ni la correction des opérations faites ;

– ce domaine au niveau du tableau est projeté sur un domaine au niveau du *template* ;

– le domaine du *template* est ensuite projeté sur le domaine de l'arrangement de *processors* correspondant, qui permettra de générer les gardes nécessaires aux échange de messages, pour l'envoi et la réception ;

– les messages dont le domaine des processeurs concernés est vide sont retirés ;

– enfin à l'aide de la description du destinataire du message, et connaissant le méthode de linéarisation des arrangements de *processors* employé par `hpfc` pour gérer les processus avec PVM, on génère le décalage sur l'arrangement linéarisé auquel correspond le destinataire.

On obtient à ce stade une description du point de vue de l'envoyeur des messages qui devront être échangés, avec le contenu du message, les processeurs concernés par l'envoi, et une description du destinataire relatif du message. À cette phase l'ensemble des messages est bien défini et on peut appliquer les transformations suivantes à cette liste :

– retirer les messages qui sont envoyés plusieurs fois, ou bien qui sont contenus dans un autre message (plus grand, selon un ordre partiel des messages fonction de leur contenu et l'ensemble des envoyeurs) ;

– regrouper les messages qui peuvent l'être (exemple, si un même ensemble de processeurs doit envoyer les messages `b(10,1:9)` et `b(10,2:10)`, autant envoyer le message `b(10,1:10)`) ; (*non implémenté*)

- pour être tout à fait général, on doit chercher tous les messages d'intersection non vide et chercher à les découper de manière à envoyer une et une seule fois la partie commune ; une telle procédure générale est cependant un peu complexe si l'on considère à la fois le domaine des processeurs et le contenu du message à envoyer ; si deux messages ont même destinataire relatif, et concerne un même tableau, une analyse fine doit être faite ; (*non implémenté*)

- après cette restructuration des messages, on peut chercher à regrouper les messages ayant même destinataire ; en toute logique, cette phase doit éventuellement redécouper les domaines de processeurs lorsque le destinataire relatif est le même et que les domaines on une intersection non vide ; cette phase est l'aggrégation des messages. (*non implémenté*)

Enfin à partir des descriptions des messages à envoyer on peut générer les messages du point de vue du receveur :

- le contenu du message doit être stocké sur les « bords » du tableau local ;

- l'ensemble des processeurs concernés par la réception, sous les hypothèses décrites, est simplement le translaté des processeurs envoyeurs par le vecteur de voisinage des destinataires ;

- le vecteur de provenance est l'opposé du vecteur des destinataires.

On génère ensuite les communications avec les gardes minimales nécessaires. Une factorisation des gardes pourrait être envisagée. Les envois et les réceptions correspondantes des messages doivent être faits dans le même ordre pour garantir la correction des communications à travers PVM. Un système de numérotation des messages est employé par `hpfc` pour assurer cette correction.

4.3.2.5 Génération du code

Le code généré remplace l'ancienne boucle, en assurant les parcours directement sur les données locales. Il suppose que les zones de recouvrement sont disponibles. Une optimisation à envisager serait de scinder cette boucle en deux parties, l'une pour les itérations dont toutes les données sont locales, à insérer entre les envois et les réceptions, et une second partie pour les itérations nécessitant des accès à distance, à placer après la réception des messages, ce qui permettrait de recouvrir les communications au maximum à l'aide de calculs.

L'implémentation comporte notamment :

- la génération les nouvelles bornes de boucle du parcours local ;

- le calcul de la valeur de l'index global si cette valeur est nécessaire au corps de boucle ;

- le calcul pour chaque référence les nouveaux indices en fonction des nouveaux index de boucle générés ;

- la génération de la garde éventuelle nécessaire sur la boucle.

- la correction des déclarations en fonction des recouvrements acceptés dans les divers analyses de boucles.

Chapitre 5

Réalisation : `hpfc`

5.1 Contexte

L'implémentation du prototype de compilateur HPFC (*High Performance Fortran Compiler*) a été faite dans le paralléliseur automatique Pips.

5.1.1 Pips

Pips (*Paralléliseur Interprocédural de Programmes Scientifiques*), est un prototype de paralléliseur automatique de Fortran qui a été principalement développé par le Centre de Recherche en Informatique de l'École des mines de Paris.

5.1.1.1 Présentation de Pips

L'arrivée de nouvelles machines à architecture parallèle sur le marché ces dernières années, l'existence de nombreux codes pour machines séquentielles représentant des investissements très importants, le désir d'améliorer les performances de ces programmes en les portant sur ces nouvelles machines ont tôt fait de pousser la recherche sur les problèmes de parallélisation automatique ou assistée, ce dès la fin des années 70. Cette opération s'était en effet montrée particulièrement difficile et coûteuse à la main, compte tenu du modèle de programmation généralement utilisé, l'échange de messages (*message passing*), qui oblige le programmeur à travailler avec des données morcelées, et à un niveau assez bas.

Le but des paralléliseurs développés à partir de ces recherches est, au moyen d'analyses sémantiques des programmes, de détecter les parties de code parallèles ou les transformations de programme qui permettraient de rendre certaines parties parallèles. De nombreux projets de recherche, voire même commerciaux, ont été lancés pour assister les programmeurs dans cette tâche difficile. Pips, le *Paralléliseur Interprocédural de Programmes Scientifiques* du Centre de Recherche en Informatique de l'École des mines de Paris, fait partie de ces outils.

Pips est un paralléliseur expérimental source à source Fortran qui a été développé depuis 1988 grâce à des fonds de la DRET (Direction des Recherches, Études et Techniques), organisme qui dépend de la DGA (Délégation Générale pour l'Armement). Son originalité est de générer et d'utiliser des informations interprocédurales dans les processus d'analyses et de parallélisation. La richesse des analyses proposées et la structure modulaire et souple du programme en font une base intéressante pour d'autres développements, implémentation d'algorithmes nouveaux pour la parallélisation ou la vectorisation, voire d'autres fins, comme la compilation d'un nouveau langage de programmation à parallélisme de données basé sur

Fortran. Le programme HPFC (*High Performance Fortran Compiler*) s'est naturellement inséré parmi les nombreux modules de Pips, ce qui a permis de profiter des nombreuses analyses disponibles, des interfaces de sorties et de l'analyseur syntaxique.

5.1.1.2 Newgen

Le paralléliseur Pips est un programme conséquent (80 000 lignes environ!) essentiellement écrit en C. À la base des structures de données manipulées par Pips se trouve l'outil de génie logiciel Newgen, développé par Jouvelot et Triolet [JT90], qui permet une description simple et riche des types de données (structures, unions, listes, tableaux, ensembles, tables de hachage, etc.). À partir de ces descriptions, Newgen génère de nombreuses fonctions pour initialiser, utiliser, détruire ou sauvegarder les données.

```
siecle  = XVI:unit + XVII:unit + XVIII:unit + XIX:unit + XX:unit ;
auteur  = nom:string x siecle ;
tableau = titre:string x auteur ;
musee   = nom:string x tableau* ;
```

Exemple de déclarations de domaines Newgen

Dans cet exemple, `siecle` est un élément parmi les 5 `unit` possibles, qui forment une union (`+`). `auteur` est une structure (`x`) à deux champs, dont un est de type `string`. Un `musee` est composé de son `nom` et d'une liste (`*`) de `tableau`.

À partir d'une telle description, Newgen génère donc pour chaque domaine `xxx` des fonctions pour initialiser (`make_xxx`), détruire (`free_xxx`), définir ou utiliser des champs `xxx_yyy`, et lire ou écrire les structures (`write_xxx`), (`read_xxx`). Dans le cas des unions, on peut tester la champs `tag` pour connaître l'objet contenu dans la structure. Des prédicats de tests sur ces champs sont également générés. En plus de ces nombreuses fonctions typées, on trouve également dans Newgen des bibliothèques de fonctions génériques pour manipuler des tables de hachage ou des listes. De plus, une fonction permet de vérifier la consistance de la structure de données, ce qui s'avère bien utile pour la recherche des erreurs dans les programmes complexes.

```
auteur
  rubens = make_auteur("Rubens",
                       make_siecle(is_siecle_XVI, UU));
tableau
  t1 = make_tableau("Le Tournoi ", rubens),
  t2 = make_tableau("La Kermesse ", rubens);
musee
  louvre = make_musee("Louvre ",
                      CONS(TABLEAU, t1, CONS(TABLEAU, t2, NIL)));
```

Exemple d'utilisation de domaines Newgen

L'utilisation (voir exemple) de cet outil dans tous les développements de Pips donne aux codes une certaine homogénéité qui facilite leur lecture. Ceci permet une meilleure réutilisabilité des codes. Cependant le déboguage des programmes n'est pas facilité, car ces structures de données ne sont pas supportées par les débogueurs disponibles, ce qui oblige à descendre à un assez bas niveau pour examiner les données pendant cette phase toujours trop longue de mise au point...

5.1.1.3 Structure de Pips

Les nombreux modules ou librairies qui composent Pips utilisent donc pour la description de leurs données l'outil de génie logiciel Newgen. Ces bibliothèques, fort nombreuses et très diverses, produisent et consomment les différentes analyses proposées par Pips pour la parallélisation automatique. En voici quelques exemples, qui soulignent la diversité des programmes qui composent Pips :

- `syntax` fait l'analyse syntaxique d'un programme Fortran proposé à la sagacité du paralléliseur ;
- `effects` calcule les effets sur les différentes variables des instructions du programme analysé ;
- `flint` fait des vérifications de correction syntaxique et sémantique sur un programme.

Chaque librairie produit et/ou utilise des ressources, dont la gestion et la permanence est assurée par `pipsdbm`, une base de données intégrée à Pips.

Les dépendances producteur-consommateur de ressources (*i.e.* d'analyses de programme ou de parties de programme proposées par les différents modules de Pips) sont gérées par l'utilitaire `pipsmake`, qui organise de façon automatique les différentes phases de manière a assurer le respect des contraintes de précédence entre elles. Ces contraintes sont déclarées dans un fichier de configuration, baptisé fort judicieusement `pipsmake.rc`.

5.1.2 Intégration de HPFC dans l'environnement Pips

5.1.2.1 Environnement de Développement

L'importance et la complexité du projet Pips, et sa structure essentiellement ouverte, ont conduit à la définition et à l'utilisation d'outils d'aide au développement, qui impliquent eux-mêmes une méthodologie et une organisation particulière. L'addition d'une nouvelle librairie dans cet environnement complexe n'est certes pas simple, mais elle permet de profiter des nombreux outils développés d'une part, et des analyses déjà disponibles d'autre part. C'est pourquoi Pips sert de base de développement à des projets connexes, comme de nouveaux algorithmes de parallélisation, ou, dans le cas présent, un prototype de compilateur pour un nouveau langage de programmation à parallélisme de données basé sur Fortran.

Pips se présente comme un répertoire, `Pips`, qui se trouve sur le compte `pips` au CRI. Dans ce répertoire, on trouve :

- `Documentation`, la documentation, c'est à dire les multiples rapports remis à la DRET lors des différentes phases de développement du projet Pips ; les déclarations Newgen des structures de données utilisées dans Pips sont tirées de leurs commentaires ;
- `Development`, la version de Pips justement en cours de développement, sur laquelle chacun fait ses interventions ;
- `Production`, une version temporairement considérée comme validée ;
- `Externals` contient les bibliothèques considérées comme externes à Pips, mais qui sont utilisées par le programme, comme `Newgen`, ou la bibliothèque de calcul linéaire `Linear` ;

– dans `Utilities` se trouvent enfin l'ensemble des scripts en shell qui facilitent le développement et la maintenance dans cet environnement : recompilation, nettoyage de l'arborescence, analyse des librairies, sauvegarde du projet, outils de validation, génération *quasi* automatique des `Makefile` des librairies, etc., pratiquement toutes les tâches répétitives inhérentes à un grand projet on été automatisées ; ces utilitaires rendent la vie du développeur Pips un peu plus facile...

HPFC, le *High Performance Fortran Compiler*, s'est donc naturellement intégré parmi les librairies de Pips. Le répertoire (`hpfc`) de ce module a été placé dans la version en cours de développement de Pips (`Pips/Development/Lib/`).

5.1.2.2 Newgen

Le compilateur utilise un certain nombre de structures Newgen nouvelles, pour retrouver les déclarations–HPF liées aux objets manipulés par le programme compilé. Ces déclarations sont, pour les directives et pour l'analyse syntaxique des directives :

```
import entity from "ri.newgen" ;
import expression from "ri.newgen" ;

align =  alignment* x template:entity ;
alignment = arraydim:int x templatedim:int x
                           rate:expression x constant:expression ;
distribute = distribution* x processors:entity ;
distribution = style x parameter:expression ;
style = none:unit + block:unit + cyclic:unit ;
```

Domaines `Newgen` pour le placement HPF

```
import distribution from "hpf.newgen" ;
import entity from "ri.newgen" ;
import expression from "ri.newgen" ;
import range from "ri.newgen" ;

alignsrc = star:unit + colon:unit + dummy:string ;
alignsubs = star:unit + range + subsuse ;
arrinalign = array:entity x alignsrc* ;
subsuse = rate:expression x dummy:string x constant:expression ;
tempinalign = template:entity x alignsubs* ;
tempindist = template:entity x distribution* ;
```

Domaines `Newgen` pour l'analyse des directives de placement HPF

Ces dernières déclarations décrivent les structures qui permettent de remonter au cours de l'analyse syntaxique les différents paramètres des déclarations–HPF tirées des directives en commentaires du programme, et de générer ces déclarations sous bonne forme en utilisant la première structure. En plus de ces déclarations de type Newgen, les types `entity_mapping` et `control_mapping` qui correspondent à des tables de hachage ont été constitués, avec toutes les fonctions nécessaires à leurs initialisations, destructions et gestions.

5.1.2.3 Dépendances entre les librairies

En plus des librairies qui proposent de manière systématique des fonctions pour aider à la programmation dans l'environnement de Pips, telles `ri-util`, `text-util` ou `misc`, HPFC utilise également des fonctions offertes par d'autres librairies :

- HPFC comprend un module d'analyse lexicale et syntaxique des directives–HPF, qui a été adapté à partir du module `syntax` de Pips, et qui réutilise un certain nombre des fonctions de ce module ;
- les vérifications des conditions de validité des différentes transformations et opérations faites au cours de la compilation s'appuient sur l'analyse des effets de Pips, dont les fonctions se trouvent dans le module `effect` ;
- la sortie des résultats entre les différentes phases de la compilation fait appel aux fonctions de `prettyprint` ;
- enfin l'accession aux différentes ressources nécessaires se fait par l'intermédiaire de la base de données de Pips, `pipsdbm`.

5.1.3 Survol de HPFC

Le prototype de compilateur HPFC (*High Performance Fortran Compiler*) a été développé dans l'environnement Pips au cours du stage. Il représente environ 10 000 lignes de C, ce qui en fait un programme conséquent. Les fonctions utilisées pour la résolution dynamique des accès aux variables réparties, écrite en Fortran 77, comptent elles 1 000 lignes de code.

5.1.3.1 Objectifs

L'objectif du prototype de compilateur HPFC est de produire un code 2PMD (*2 Programs, Multiple Data*) pour Fortran 77 additionné d'une librairie d'échange de messages, à partir d'un sous-ensemble raisonnable mais néanmoins large de HPF.

Le sous-ensemble considéré comprend Fortran 77 avec les directives statiques de placement des données, `ALIGN` et `DISTRIBUTE`, ainsi que la directive de parallélisme `INDEPENDENT`. Les problèmes d'appels de fonctions ou routines avec des tableaux répartis en paramètre ne sont cependant pas traités, cela ayant été jugé hors de portée sur la durée impartie au stage.

Le modèle de machine cible (voir figure 5.1) choisi comprend un processeur spécialisé `host` pour les entrées-sorties, et un certain nombre de processeurs nœuds `node`, tous ces processeurs pouvant communiquer arbitrairement entre eux. L'un des programmes produits est destiné au processeur `host`, et l'autre est un programme SPMD pour les nœuds `node`.

Pour éviter d'avoir à gérer un niveau supplémentaire entre les données et les processeurs, les éléments des arrangements de *processors* ont été identifiés à des processus, et il tournera autant de processus que de *processors* déclarés dans le programme en HPF. Cette identification n'est pas sans influence sur les performances éventuelles du programme pour un nombre plus réduit de processeurs, chacun recevant éventuellement plusieurs processus :

- sur un même processeur est alors exécuté plusieurs fois le flot de contrôle du programme, ce qui est très pénalisant pour le procédé de résolution dynamique des accès, puisque chaque processus parcourt déjà le flot de contrôle total à la recherche de ses interventions éventuelles ;

- cependant ce fait peut être parfois bénéfique lorsque des processus deviennent libres (*idle*), en attente de l'arrivée de messages, et le temps cpu dégagé peut être avantageusement utilisée par un autre processus.

Le code généré s'appuie sur des fonctions et routines de communication, mais aussi d'examen de la localité des données souhaitées, qui sont elles-mêmes programmées en utilisant les fonctions de communication fournies par PVM et la description des objets répartis générée par le compilateur. Ces différentes fonctions utilisent et définissent des données qui se trouvent dans des `COMMON` Fortran.

La description du travail effectué par le compilateur est décrite aux deux sections suivantes, dont voici brievement un description du contenu.

5.1.3.2 Compilation

L'algorithme de réécriture $\mathcal{T}_0$ ne peut être directement appliqué sur le code analysé par Pips. Il faut auparavant :

- **analyser** les directives HPF de placement des données ;
- **normaliser** le code analysé de manière à imposer les conditions nécessaires à la validité de $\mathcal{T}_0$, ce qui se traduit principalement par la réduction des indirections sur les variables réparties du programme ;
- redéfinir de nouvelles **déclarations** pour les tableaux répartis de manière à réduire autant que possible l'espace réservé sur chaque nœud pour contenir les morceaux de tableaux.

Ces phases accomplies, l'algorithme de réécriture $\mathcal{T}_0$ peut être appliqué sur le code, pour produire les 2 programmes désirés ainsi que les routines d'initialisation des descriptions.

5.1.3.3 Exécution

Le code produit par $\mathcal{T}_0$ s'appuie des fonctions de support, qui s'appuient elles-mêmes sur des données et les primitives de communications de PVM 3.1.2. Cette partie (section 5.3 décrit ces différents aspects :

- les données de support ;
- les fonctions de support ;
- les contraintes dues à l'utilisation de PVM ;

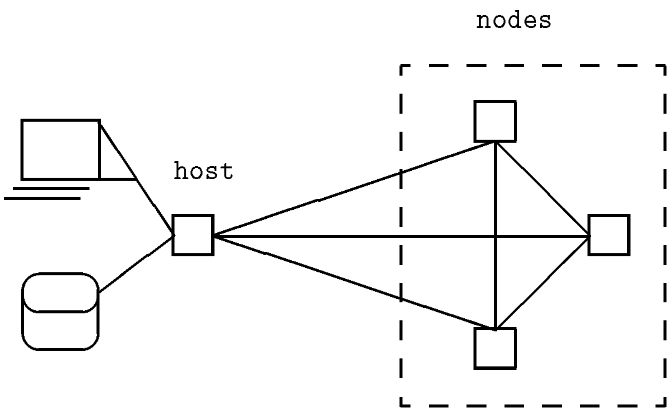

FIG. 5.1 - *Modèle de machine parallèle cible*

5.2 Compilation

L'algorithme de réécriture proposé à la section 4.2 ne peut pas être directement appliqué à la représentation du programme manipulées par Pips. Il faut auparavant :

- récupérer les directives HPF, tant en ce qui concerne la désignation des boucles parallèles que les déclarations d'alignement et de distribution ;

- **normaliser** le code, de façon à imposer les conditions nécessaires à la validité des transformations qui seront faites à la compilation ;

- générer de nouvelles déclarations pour les tableaux répartis.

5.2.1 Déclarations-HPF

Cette phase consiste en l'analyse syntaxique des directives-HPF de déclarations des alignements et distributions des objets du programme, et en la génération des structures qui à la compilation contiennent ces descriptions. Elle est assurée au moyen d'un préfiltrage des directives et d'un analyseur syntaxique dédié. On évite ainsi de perturber l'analyseur de Pips qui fonctionne et sert de base aux travaux des autres développeurs.

5.2.1.1 Préfiltrage

Le filtre utilisé est un shell qui récupère les lignes commençant par les préfixes autorisés de directives-HPF (`CHPF$`, `!HPF$` ou `*HPF$`) qui les signalent. Ce même filtre réajuste les lignes de commentaire coupées par un esperluet (`&`). On ne retient par ce filtrage que les directives de déclarations `TEMPLATE`, `PROCESSORS`, `ALIGN` et `DISTRIBUTE`, à l'exclusion de toute autre directive. Le résultat de ce filtrage est placé dans un fichier à part qui sera traité par l'analyseur.

```
#! /bin/sh
#
# this shell-script filters a Fortran program for HPF directives.

  tr    'a-z' 'A-Z'                                        \
| sed   -n 's/^[C!*]HPF\$[        ]*//p'                    \
| awk   'BEGIN{PREC = ""}                                  \
        /.*&$/{ PREC = PREC substr($0,1,length-1) }        \
        (!/.*&$/){ print PREC $0; PREC = ""}'              \
| egrep '^(TEMPLATE|PROCESSORS|ALIGN|DISTRIBUTE)'
```

Filtre pour les déclarations statiques de HPF

Le problème des directives dynamiques reste entier dans ces conditions. Il faut savoir en quels points du programme ont lieu ces réarrangements des données, et cette information est difficile à obtenir dans le cadre d'un mécanisme de préfiltrage. Pour les directives de parallélisme de boucles, la récupération de l'information se fait au niveau de la phase de normalisation du code.

5.2.1.2 Analyse syntaxique

L'analyseur syntaxique des déclarations HPF se charge ensuite d'extraire du fichier produit les informations nécessaires à la définition des structures de données internes à la phase de compilation, qui doivent contenir la description des alignements et distributions des données. La syntaxe acceptée par l'analyseur a été simplifiée de manière à accélérer le développement de cette partie du compilateur, mais la totalité de la sémantique possible des alignements et distributions est supportée, les restrictions n'ayant qu'une nature essentiellement syntaxique (section 3.3.2).

- directives `TEMPLATE` et `PROCESSORS` :

  on peut déclarer une liste d'objets, de la même manière que pour les variables de Fortran ; cependant les passes ultérieures supposent que la taille de chaque dimension de ces tableaux est statiquement connue ;

- directive `ALIGN` :

  seule la syntaxe s'appuyant sur la notation `*` et l'utilisation des variables muettes est implantée ; la nature des formes affines acceptées lors de la spécification des alignements est également réduite ; sont autorisés les constantes, l'identité et la forme `a*I+b` avec `a` et `b` entiers ; sont donc exclues les appels à des fonctions intrinsèques, même statiquement évaluables ;

- directive `DISTRIBUTE` :

  le paramètre éventuel de la distribution doit être un entier, comme dans le cas de l'alignement ; pour le reste, la syntaxe est celle de la spécification HPF.

Enfin les alignments et distributions doivent impérativement avoir lieu sur des *templates* et *processors* désignés et déclarés, ce qui implique un ordre sur la prise en considération des directives. Les alignements implicites ou indirects, et les distributions implicites ou sur des objets implicites, ne sont pas supportés par cette version de l'analyseur syntaxique. Des versions ultérieures pourront éventuellement insérer les objets manquants à leur place, et calculer les paramètres résultants des alignements indirectes. De plus, le sucre syntaxique proposé, et lié à Fortran 90, n'est pas intégré à cette version de l'analyseur syntaxique.

```
      REAL A(10)
CHPF$ TEMPLATE T(10)
CHPF$ ALIGN A(I) WITH T(I)
CHPF$ PROCESSORS P(2)
CHPF$ DISTRIBUTE T(BLOCK) &
CHPF$   ONTO P
```

```
      INTEGER J(4:10,10),K(20)
CHPF$ TEMPLATE T(100,100)
CHPF$ ALIGN K(I),J(I,*) WITH &
CHPF$   T(-2*I+60,*)
CHPF$ PROCESSORS P(4,4)
CHPF$ DISTRIBUTE  &
CHPF$   T(BLOCK(50),CYCLIC) ONTO P
```

Exemples de directives analysées

Seul un sous-ensemble des directives HPF est donc accepté à cette phase, mais ce sous-ensemble comprend toute la sémantique des répartitions de données de HPF.

5.2.2 Normalisation

La phase de normalisation du code :

- réduit les indirections ;
- recherche les boucles parallèles.

5.2.2.1 Indirections

Un certain nombre de conditions sur le code sont nécessaires pour générer le code avec échange de message. Une passe assure donc les transformations impliquées pour que le code proposé en entrée du compilateur respecte ces conditions. Cette passe exécute également des transformations qui visent à faciliter la résolution dynamique.

Ces interventions sont :

- réductions des indirections de variables réparties, condition qui est nécessaire pour connaître au niveau d'une expression toutes les variables réparties lues ou écrites ;
- calcul explicite des indices d'accès aux variables réparties, pour éviter de calculer plusieurs fois une même expression au niveau de la résolution dynamique.
- les bornes de boucle des tests sont également cassées, au cas où des effets de bords auraient lieu au niveau du calcul de la condition et nécessiteraient une mise à jour de la valeur d'une variable répartie.

Par exemple, le code d'école suivant :

```
      program hpftest22
      integer j(10),k(10)
chpf$ template t(10)
chpf$ processors p(2)
chpf$ align j(i), k(i) with t(i)
chpf$ distribute t(block) onto p
      print *, 'hpftest22 running'
chpf$ independent(i)
      do i=1, 10
         k(i) = mod(i+6,10)+1
         j(i) = 11-i
      enddo
      do i=1, 10
         print *,'j(k(i)) = ',j(k(i))
      enddo
      if (j(k(3)).LT.j(k(7))) then
         print *, 'quite interesting a property...'
      endif
      print *, 'hpftest22 end'
      end
```

sera transformé ainsi :

```
      PROGRAM HPFTEST22
      INTEGER INT_HPF_2,INT_HPF_1,INT_HPF_0,J(1:10),K(1:10),I
      LOGICAL LOGICAL_HPF_0
      WRITE (UNIT=6,FMT=*) 'HPFTEST22 RUNNING'
CHPF$ INDEPENDENT(I)
      DOALL I = 1, 10
         K(I) = MOD(I+6, 10)+1
         J(I) = 11-I
```

```
ENDDO
DO I = 1, 10
   INT_HPF_0 = K(I)
   WRITE (UNIT=6,FMT=*) 'J(K(I)) = ',J(INT_HPF_0)
ENDDO
INT_HPF_2 = K(7)
INT_HPF_1 = K(3)
LOGICAL_HPF_0 = J(INT_HPF_1).LT.J(INT_HPF_2)
IF (LOGICAL_HPF_0) THEN
   WRITE (UNIT=6,FMT=*) 'QUITE INTERESTING A PROPERTY...'
ENDIF
WRITE (UNIT=6,FMT=*) 'HPFTEST22 END'
END
```

Après le passage de la phase de normalisation, on constate que le code a, au moins dans cet exemple, quelques propriétés recherchées :

- la boucle déclarée parallèle dans le code source, avec la directive `INDEPENDENT`, a été marquée parallèle (`DOALL`) ;
- les indices d'accès à des variables partagées sont explicitement calculés ;
- il n'y a pas plus d'indirections de variables réparties.

Il manque encore le traitement des boucles implicites dans les Entrées-Sorties sur variables réparties.

5.2.3 Boucles parallèles

Comme l'exemple ci-dessus le montre, les boucles parallèles sont détectées lors de la phase de normalisation. La procédure recherche parmi les commentaires liés aux boucles l'invocation magique de la directive `INDEPENDENT`, puis marque comme parallèles les boucles dont les indexes sont désignés par la-dite directive.

5.2.4 Traitement des tableaux

Les tableaux répartis sur les différents processeurs nécessiteront éventuellement une place moindre sur chaque nœud de la machine cible. Pour chaque dimension de chaque tableau réparti, en fonction des alignement et distribution qu'elle suit, la nouvelle taille est calculée, en majorant l'espace nécessaire sur le nœud le plus chargé (modèle 2PMD avec déclarations statiques). Ainsi, dans le cas d'un tableau aligné et distribué sur un seul processeur, on est conduit à allouer la place du tableau sur tous les processeurs.

5.2.4.1 Re-déclarations

Les nouvelles déclarations des tableaux distribués doivent permettrent de réduire l'espace mémoire nécessaire sur les nœuds d'une part, mais aussi de calculer les nouveaux indices locaux à partir des valeurs des indices globaux de manière aussi simple que possible.

De plus, certaines optimisations envisagées consisteront à parcourir directement les tableaux locaux pour y faire les opérations demandées. Il faut donc que la structure des données locales se prête à ce genre d'optimisations.

Dans le cas des déclarations suivantes, en considérant les dimensions des tableau et *processors*:

objet	*taille*	*bornes*	
tableau	s	a_m	a_M
processors	p	p_m	p_M

on propose de redéclarer la taille de chaque dimension du tableau en fonction des ses alignement et distribution, de manière à, autant que faire ce peu, minimiser l'espace mémoire nécessaire au tableau sur chaque processus :

cas	alignement	distribution	re-déclaration
α	-	-	$1:s$
β	$ai+b, a\neq 0$	`block(`n`)`	$1:\lceil\frac{n}{\lvert a\rvert}\rceil$
γ	$i+b$	`cyclic(`n`)`	$1:n\lceil\frac{s+n-1}{np}\rceil$
δ	$ai+b, a\neq 0$	`cyclic(`n`)`	$1:\lceil\frac{n}{\lvert a\rvert}\rceil\lceil\frac{\lvert a\rvert(s-1)+n}{np}\rceil$

Si plusieurs formules conviennent, celle conduisant à la déclaration la plus réduite est choisie. En pratique, on compare donc la taille proposée par les formules β, γ et δ avec la taille initiale du tableau, qui correspond au cas α.

5.2.4.2 Calcul des nouveaux indices

On peut alors calculer les indices locaux i', en fonction de l'indice global i et des différents paramètres des déclarations, alignements et distributions :

cas	résultat
α	$i' = i - a_m + 1$
β	$i' = \frac{(ai+b-t_m)\bmod n}{\lvert a\rvert} + 1$
γ	$\begin{cases} s_c = np \\ n_0 = \frac{a_m+b-t_m}{s_c} \\ t = i+b-t_m \\ n_c = \frac{t}{s_c} \\ n_i = t \bmod n \\ i' = n(n_c - n_0) + n_i + 1 \end{cases}$
δ	$\begin{cases} s_c = np \\ n_0 = \frac{aa_m+b-t_m}{s_c} \\ t = ai+b-t_m \\ n_c = \frac{t}{s_c} \\ n_i = \frac{t \bmod n}{\lvert a\rvert} \\ i' = \lceil\frac{n}{\lvert a\rvert}\rceil(n_c - n_0) + n_i + 1 \end{cases}$

Noter que dans le cas de la déclaration γ, on peut majorer l'espace mémoire localement nécessaire par

$$n\lceil\frac{s}{np}\rceil$$

mais la formule de calcul de l'indice local est dans ce cas un peu plus complexe (un modulo est ajouté), ce qui n'est pas souhaité. Cette déclaration à la propriété intéressante de placer de manière contigue et ordonnées les éléments de tableaux placés sur un même processeur, propriété qui sera utile dans les phases ultérieures d'optimisation, lorsque des parcours de ces éléments devront être faits.

De plus, les deux premières formules du calcul de l'indice local peuvent être calculées statiquement, au moment de la compilation. De manière à accélérer l'exécution de la résolution dynamique des accès du programme, les paramètres de ces différentes formules de calcul des indice locaux d'une dimension sont dans la mesure du possible calculés statiquement (voir 5.3.1).

On peut envisager de traiter plus finement les calculs d'indices locaux, et de faire des formules adaptées à certains cas particulier, comme $\beta_{a=1}\ldots$

5.2.5 Algorithme $\mathcal{T}_0$

5.2.5.1 Réécriture

L'application de l'algorithme $\mathcal{T}_0$ pour la compilation est une double réécriture de l'arbre syntaxique du programme normalisé pour produire les programmes désirés en fonction des règles décrites à la section 4.2.

Cette réécriture consiste à parcourir l'arbre syntaxique en allant de la racine vers les feuilles (*Top down*), de vérifier à chaque nœuds quelles sont les conditions des sous arbres, à ce niveau, et d'orienter sur la bonne phase de compilation en fonctions des différentes conditions. Deux modules sont créés au cour de la réécriture, nommés `NODE` et `HOST`, et qui contiennent les deux programmes résultant de la compilation.

5.2.5.2 Optimisations

Un certain nombre d'optimisation de base ont été implémentées dans l'algorithme, qui permettent d'améliorer un peu le code produit :

- un algorithme d'analyse des recouvrements, décrit en 4.3, a été implémenté, ce qui permet lorsque les conditions sont réunies d'améliorer les performances de l'exécution.
- on sait statiquement si les données d'un tableau réparti sont dupliquées ou non, et l'appel aux fonctions de mise à jour n'est pas fait lorsque cela n'est pas nécessaire ;
- les communications sont également simplifiées dans le cas où il n'y a pas de duplication ; des appels à des fonctions plus simples sont alors générés ; cela permet de gagner un certain temps par rapport aux fonctions générales ;
- les calculs des indices locaux ont été insérés directement dans le code, et les expressions simplifiées lorsque cela était possible, afin de rendre le code produit un peu plus compréhensible et lisible ;
- les boucles vides générées éventuellement par certaines phases de compilation sont retirées ;
- une optimisation qui pourrait être ajoutée consiste à regrouper les accès aux mêmes variables avant de générer les communications, contrairement à ce qui est fait actuellement ; une telle optimisation est liée à une bonne gestion des effets calculés sur les instructions.

5.2.6 Bogues, Dysfonctionnements et Lacunes

Le développement rapide du *High Performance Fortran Compiler* a laissé un certain nombre de bogues, de dysfonctionnements et de lacunes dans le programme, que ce soit dans le langage accepté en entrée, ou même dans ce qui est généré. Voici donc une petite liste de ce qui ne fonctionne pas ou mal dans `hpfc`.

Fortran 77 : le langage d'entrée du compilateur est Fortran 77, qui est un sous ensemble de Fortran 90, le langage de base de HPF. Même par rapport au *subset* officiel de HPF, il manque des traits tout à fait essentiels du langage, comme la notation des sections de tableaux (*array sections*), les fonctions intrinsèques de réduction, ou la construction à parallélisme de données `WHERE`.

Forall : l'instruction `forall` n'est également pas analysée par l'analyseur syntaxique, et n'est donc pas traitée, malgré son caratère fondamental dans HPF, principalement pour les machines à flot unique de données (SIMD).

Boucles Indépendantes : la directive `INDEPENDENT` n'est pas bien gérée : récupération laborieuse des index de la boucle, et le complément sur les variables privées au corps de boucle n'est pas pris en compte.

Appels de fonctions : l'aspect procédural du langage n'est également pas considéré, et toutes les fonctions sont traitées comme si elles étaient sans effets de bords. De plus, il n'est pas possible d'appeler des sous-programmes en passant en paramètres des tableaux répartis, comme cela est autorisé en HPF. Une version ultérieure devra au moins permettre de traiter les appels de sous=programme avec des paramètres modifiés.

Écarts des directives avec HPF : un certain nombre de simplifications ont été faites au niveau des directives acceptées par le compilateur, qui ne réduisent pas la sémantique traitée par HPFC, sous la condition que les valeurs des paramètres de ces directives (tailles, bornes, coefficients) soient accessibles au moment de la compilation. Ces écarts sont :

- pas de déclaration implicite des objets ou correspondances : les directives HPF peuvent s'appuyer sur des objets implicites qui ne sont donc pas explicitement déclarés et manipulés (alignement entre tableaux, arrangement de processeur non spécifié dans une distribution, etc.) ; le compilateur ne gère pas ce genre de problèmes, c'est à dire qu'il ne normalise pas les déclarations ; ils faut donc que celles-ci soient directement de la forme attendue, alignements sur un *template* déclaré et distribution sur un arrangement de *processors* ;
- syntaxe réduite des alignements : les alignements peuvent être spécifiés avec plusieurs syntaxes, la première basée sur les sections de tableaux, et la seconde sur la spécification de fonctions affines avec des indices muets variant entre les bornes de la dimension du tableau concerné. Seule la seconde syntaxe est pour l'instant autorisée, mais l'analyseur syntaxique accepte les deux, et il manque juste la fonction génération de la description à partir des informations disponibles dans le premier cas. Toutes les expressions affines ne passent cependant pas l'analyseur, mais tout peut être exprimé, ce qui est l'essentiel dans cette phase de développement préliminaire d'un prototype.

- limitation des valeurs acceptées : seules les valeurs entières explicites sont gérées par le compilateur, comme paramètres des alignements et distributions, et pour les tailles et bornes des objets déclarés et manipulés, comme les tableaux répartis, les *templates* et les *processors* ; HPF permet normalement l'utilisation de constantes symboliques, et même l'appel à des fonctions intrinsèques, de calcul mais aussi liés à HPF, comme `NUMBER_OF_PROCESSORS()` ; tout cela n'est pas pris en compte ;
- `COMMONS` : ils ne peuvent pas être répartis dans le version actuelle du compilateur, même si cela ne devrait pas poser plus de problèmes que dans le cas des tableaux normaux ; les alignements en question ne sont pas analysés.

Entrées-Sorties : le cas des boucles implicites dans les entrées-sorties n'est pas pris en compte par le compilateur, et en conséquence les seules entrées-sorties possibles sont sur des scalaires ou des éléments de tableaux. Cela devrait être corrigé, éventuellement avec des messages regroupés quand c'est possible.

Effets : les effets considérés pour les fonctions sont inexacts, et toutes les fonctions sont considérées comme sans effet de bords (leur paramètres sont considérés comme uniquement lus). La seule manière de définir une variable qui sera compilée de manière correcte est l'affectation directe de cette variable. Cette restriction devra être levée par une bonne exploitation du calcul des effets de Pips, qui n'a pas été faite pour des raisons diverses exposées en 7.1.1.

Boucles : les boucles dont le corps est vide sont directement retirées des codes produits, sans prendre garde à l'utilisation éventuelle de la valeur de l'indice à la fin de la boucle pour une raison ou une autre. Dans ce dernier cas, la sémantique du programme n'est certainement pas conservée. Ceci est donc un bogue à corriger.

Variables générées dans le mauvais module : les variables qui apparaissent dans les modules `host` et `node` sont normallement déclarée dans ces modules, et les statements de ces modules sont donc normallement corrects... sauf en ce qui concerne les générations d'appels aux fonctions de calcul des positions locales, définies parmi les fonctions de support. Ceci est encore un bogue à corriger à l'occasion, si le besoin s'en fait sentir, par exemple si on veut analyser avec Pips les codes produits directement, sans passer par une nouvelle analyse syntaxique.

Conditions non vérifiées : les conditions nécessaires à la compilation des boucles parallèles selon la forme définie dans la description de $\mathcal{T}_0$ ne sont pas vérifiées, et si il se trouve qu'elles ne sont pas réunies, la compilation donne un résultat faux. La principale contrainte qui existe pour l'instant est que les accès aux valeurs des variables réparties doivent être *atomiques*, c'est à dire que l'on doit connaitre au moment de l'entrée dans la boucle l'ensemble des accès qui seront fait. Mais l'analyse faite ne vérifie pas les éventuels boucles internes qui pourraient induire un parcours de vecteur plutôt que le simple accès à un élément, d'où l'erreur.

5.3 Exécution

5.3.1 Données de Support

Un certain nombre de données sont nécessaires pour résoudre dynamiquement les accès aux variables réparties. Ces données sont paramétrées de manière à ajuster leurs déclarations à ce qui est nécessaire. Ce paramétrage est décrit en 5.3.1.1. On a ensuite une présentation des variables globales qui sont utilisées à l'exécution, et qui ont été scindées en trois `COMMON` Fortran :

`COMMON`	*contenu*
`HPFC_PARAM`	description des objets
`HPFC_PROCS`	description des processeurs
`HPFC_GLOBS`	désignation des objets manipulés

5.3.1.1 Paramètres

Les paramètres sont générés automatiquement par le compilateur, et décrivent simplement le nombre d'objets de chaque catégorie ainsi que la taille du plus grand des *processors* déclaré dans le programme. Ces paramètres permettent d'ajuster les déclarations des variables qui servent à la résolution dynamique. Un exemple de fichier de paramètres généré est présenté à la figure 5.2. Ces paramètres sont :

nom	*contenu*
`INTFLAG`	*not a number*
`NBOFARRAYS`	nombre de tableaux
`NBOFTEMPLATES`	nombre de *templates*
`NBOFPROCESSORS`	nombre de *processors*
`MAXSIZEOFPROCS`	taille du plus grand *processors*

5.3.1.2 Description des Objets

Les objets manipulés sont décrits dans un `COMMON` Fortran baptisé `HPFC_PARAM`, et présenté à la figure 5.3. Chaque objet manipulé dynamiquement est numéroté selon sa classe, tableau, *template* ou *processors*, à partir de 1 et jusqu'au nombre d'objet de la classe concernée. Ce numéro connu à la compilation permet de désigner les objets lors des manipulations de la résolution dynamique, et d'accéder ainsi aux tables qui décrivent ces objets.

Pour chaque objet de type `X` (`X` dans `A` pour *array*, `T` pour *template* ou `P` pour *processors*) et de numéro i, on a les tableaux de description :

Tableaux de description des objets HPF :

variable	*contenu*
`NODIMX(`i`)`	le nombre de dimension de l'objet (0 à 7 en Fortran 77)
`RANGEX(`i`,`dim`,1...3)`	les taille et bornes de chaque dimension dim de l'objet i

Signification des indices sur la troisième dimension de `RANGEX` :

index	*contenu*
1	*lower*
2	*upper*
3	$size = upper - lower + 1$

```
c
c file parameters.h
c
c this file has been automatically generated by the hpf-compiler
c
c type declarations
c
      integer
     $     INTFLAG,
     $     NBOFARRAYS,
     $     NBOFTEMPLATES,
     $     NBOFPROCESSORS,
     $     MAXSIZEOFPROCS
c
c parameters
c
      parameter(INTFLAG = -1234567890)
      parameter(NBOFARRAYS = 1)
      parameter(NBOFTEMPLATES = 1)
      parameter(NBOFPROCESSORS = 1)
      parameter(MAXSIZEOFPROCS = 10)
c
c end of automatically generated file
c
```

FIG. 5.2 - *Fichier* `parameters.h` *généré automatiquement*

Pour les tableaux de données on a de plus :

variable	*contenu*
`ATOT(`i`)`	le numéro du *template* sur lequel le tableau i est aligné.
`ALIGN(`i`,`$dimt$`,1..3)`	les paramètres de l'alignement du tableau i pour la dimension du *template* $dimt$.

Troisième dimension de `ALIGN` :

index	*contenu*
1	dimension du tableau aligné
2	pente a
3	constante b

Signification du champs `1` :

valeur	*signification*
`INTFLAG`	pas d'alignement
0	alignement *constant*
autre	dimension

`RANGEA(`i`,`dim`,4..10)` est détenteur des différents paramètres nécessaires pour retrouver l'indice local sur la dimension dim du tableau i, après re-déclaration des tableaux. On trouve en `4` l'indicateur de la nouvelle déclaration faite :

`RANGEA(`i`,`dim`,4)` :

valeur	*re-déclaration*
0	*none*
1	α
2	β
3	γ
4	δ

`RANGEA(`i`,`dim`,5..8)` (voir 5.2.4.1) :

index	*none*	α	β	γ	δ
5	-	$1-a_m$	n	n	n
6	-	-	a	s_c	s_c
7	-	-	$b-t_m$	n_0	n_0
8	-	-	-	$b-t_m$	$b-t_m$
9	-	-	-	-	a
10	-	-	-	-	$\lceil \frac{n}{\lvert a \rvert} \rceil$

Enfin pour les *templates* on a :

variable	*contenu*
`TTOP(`i`)`	le numéro du *processors* sur lequel le *template* i est distribué.
`DIST(`i`,`$dimp$`,1..2)`	la distribution de *template* i, pour la dimension $dimp$ du *processors* considéré.

Troisième dimension de `DIST` :

index	*contenu*
1	dimension du *template* distribué
2	valeur du paramètre de distribution

Paramètre de distribution :

valeur	*signification*
< 0	`cyclic`
> 0	`block`

Une autre solution aurait pu être envisagée pour la description à l'exécution des objets, alignements et distributions, qui aurait consisté à ajouter un niveau d'indirection entre les objets et leur *range*, ce qui aurait permis d'éviter les descriptions en grande partie vide, puisqu'elles sont déclarées pour un nombre de dimension éventuel de 7. Comme Fortran 90 permet (?) plus de 7 dimensions, et qu'il est dommage d'allouer inutilement de l'espace mémoire, la prochaine version des descriptions ajoutera ce niveau d'indirection. Dans l'état actuel du prototype, la description directe a permis d'accélérer le développement et de faciliter la lecture manuelle des initialisations produites par le compilateur.

variable	*contenu*
`MYLID`	`lid` du processus
`MYTID`	`tid` du processus
`HOSTTID`	`tid` du processus `host`
`NODETIDS(`i`)`	`tid` associés au processus de `lid` i
`MYPOS(1..7, `i`)`	indices du processus dans le *processors* i
`NBOFTASKS`	nombre de tâches
`NODECHANNELS(`i`)`	nombre de communications entre le processus courant et le processus i
`HOSTCHANNEL`	...avec `host`
`MCASTHOST`	...sur le canal *multicasthost*
`MCASTNODES`	...sur le canal *multicastnode*

TAB. 5.1 - *Contenu du common* `HPFC_PROCS`

5.3.1.3 Description des Processeurs

Dans un souci de simplification, afin d'éviter d'avoir à gérer un niveau supplémentaire entre les données et les processeurs réels, les *processors* de HPF on été assimilés à la notion de *processus*. Chaque processus, que ce soit l'un des nœuds (`node`) ou bien `host`, doit pouvoir connaître le processus qui joue le rôle d'un élément d'un arrangement de *processors*. Toute la correspondance est décrite dans le `COMMON HPFC_PROCS` (voir figure 5.4 et table 5.1).

À chaque élément de *processors* a donc été associé un numéro unique pour chaque arrangement, appelé `lid` (*local identifier*). Les différents *processors* sont superposés les uns aux autres, de manière à ne pas lancer trop de processus, et le numéro `lid` d'un élément de *processors* est calculé en linéarisant l'arrangement par colonne.

À chaque `lid` correspond dans une table un `tid` (*task identifier*), qui est le numéro unique d'un processus dans l'environnement PVM, et qui permet de le désigner et de lui envoyer des messages.

Les messages PVM doivent de plus être assortis d'un numéro de préférence unique, pour assurer qu'aucun message entre deux processus ne puisse être confondu avec un autre. Ceci pourrait donner lieux à des bogues non reproductibles et difficiles à traquer. On garde donc trace des communications qui ont eu lieu sur un canal (lien virtuel entre deux processus particulier), ce qui permet de générer ce numéro unique lorsque nécessaire.

5.3.1.4 Objets Courants

Pendant l'exécution du programme, les processus mettent à jour les variables du `COMMON HPFC_GLOGS` (figure 5.5, table 5.2) qui décrivent les différents *processors* impliqués par un accès à une variable répartie. Ces variables servent à déterminer par qui, et à qui doit être envoyé un message dans les différentes fonctions de la résolution dynamique. Elles contiennent différentes vues des intervenants (*Owners*, *Sender*, *Computer*), et sont mises à jour par certaines fonctions.

```
c
c COMMON /HPFC_PARAM/
c
c the following file has to be included:
c     include 'parameters.h'
c
      common /HPFC_PARAM/
     $     ATOT(NBOFARRAYS),
     $     TTOP(NBOFTEMPLATES),
     $     NODIMA(NBOFARRAYS),
     $     NODIMT(NBOFTEMPLATES),
     $     NODIMP(NBOFPROCESSORS),
     $     RANGEA(NBOFARRAYS, 7, 8),
     $     RANGET(NBOFTEMPLATES, 7, 3),
     $     RANGEP(NBOFPROCESSORS, 7, 3),
     $     ALIGN(NBOFARRAYS, 7, 3),
     $     DIST(NBOFTEMPLATES, 7, 2)
c
      integer
     $     ATOT, TTOP, NODIMA, NODIMT, NODIMP,
     $     RANGEA, RANGET, RANGEP, ALIGN, DIST
```

Fig. 5.3 - *COMMON /HPFC_PARAM/*

```
c
c COMMON /HPFC_PROCS/
c
c the following file has to be included:
c     include 'parameters.h'
c
      common /HPFC_PROCS/
     $     MYLID,
     $     MYTID,
     $     HOSTTID,
     $     NODETIDS(MAXSIZEOFPROCS),
     $     MYPOS(7, NBOFPROCESSORS),
     $     NBOFTASKS,
     $     NODECHANNELS(MAXSIZEOFPROCS),
     $     HOSTCHANNEL,
     $     MCASTHOST,
     $     MCASTNODES
c
      integer
     $     MYLID, MYTID, HOSTTID, NODETIDS, NBOFTASKS, MYPOS,
     $     NODECHANNELS, HOSTCHANNEL,  MCASTHOST, MCASTNODES
```

Fig. 5.4 - *COMMON /HPFC_PROCS/*

```
c
c COMMON /HPFC_GLOBS/
c
c the following file has to be included:
c     include 'parameters.h'
c
      common /HPFC_GLOBS/
     $     OWNERTID,
     $     OLID,
     $     OPN,
     $     OINDP(7),
     $     OREPLICATED,
     $     SENDERTID,
     $     SLID,
     $     COMPUTERTID,
     $     CLID,
     $     CPN,
     $     CINDP(7),
     $     CPOS(7),
     $     CPOSPN,
     $     CPOSCOMPUTED,
     $     NLID,
     $     NTID
c
      integer
     $     OWNERTID,    OLID, OPN, OINDP,
     $     SENDERTID,   SLID,
     $     COMPUTERTID, CLID, CPN, CINDP, CPOS, CPOSPN,
     $     NLID, NTID
c
      logical
     $     OREPLICATED, CPOSCOMPUTED
```

FIG. 5.5 - *COMMON /HPFC_GLOBS/*

variable	*contenu*
OWNERTID	tid du *Owner*
OLID	lid du *Owner*
OPN	*processors number* du *Owner*
OINDP(7)	coordonnées du *Owners*
OREPLICATED	booléen *vrai si duplication*
SENDERTID	tid du *Sender*
SLID	lid du *Sender*
COMPUTERTID	tid du *Computer*
CLID	lid du *Computer*
CPN	*processors number* du *Computer*
CINDP(7)	coordonnées du *Computer* dans CPN
CPOS(7)	coordonnées du *Computer* dans CPOSPN
CPOSPN	*processors number* relatif à CPOS
CPOSCOMPUTED	booléen *vrai si CPOS... à jour*
NLID	lid du *neighbour* courant
NTID	tid du *neighbour* courant

TAB. 5.2 - *Contenu du common HPFC_GLOBS*

5.3.2 Fonctions de Support

L'objectif de la compilation est de générer un code avec échange de messages, en utilisant les fonctions de la librairie PVM 3.1 distribuée par l'ORNL (*Oak Ridge National Laboratory, Tennessee*) gratuitement. Cette librairie est constitués de fonctions de base pour échanger des messages entre processus de machines sous Unix, configurées en un réseau hétérogène par PVM. L'utilisation de cette librairie permet d'envisager une bonne portabilité du code produit par le compilateur HPFC.

Le code ne s'appuie pas directement sur PVM, mais sur des fonctions dont les spécifications suivent et qui sont, elles, écrites en utilisant cette librairie. Ces fonctions utilisent évidemment les descriptions des alignements et distribution (voir 5.3.1). Elles définissent et s'appuient sur des variables globales qui contiennent à tout moment, de manière directe ou indirecte, les *Sender*, *Owners* et *Computer* courant des données manipulées.

Calcul à l'exécution des processus impliqués par un accès à des variables réparties, et mise à jour des descriptions correspondantes dans le COMMON HPFC_GLOBS:

subroutine:

nom	*fonction*
HPFC_CMPCOMPUTER(i, ...)	mise à jour de la description du *Computer* courant (*Sender* de la variable de numéro i dont les indices suivent)
HPFC_CMPOWNER(i, ...)	mise à jour de la description des *Owners* du *Sender* courant, pour la variable en paramètre.
HPFC_CMPNEIGHBOUR(d)	mise à jour du *neighbour* courant

Prédicats pour les différents tests sur les variables dont la description est présente dans le `COMMON HPFC_GLOBS`:

`logical function`:

nom	*fonction*
`HPFC_SENDERP()`	vrai si *Sender*
`HPFC_OWNERP()`	vrai si $\in$ *Owners*
`HPFC_COMPUTERP()`	vrai si *Computer*
`HPFC_COMPUTERINOWNERSP()`	vrai si *Computer* $\in$ *Owners*

Envoi de messages à destination d'un ou plusieurs processus, selon les besoins et les descriptions courantes :

`subroutine`:

nom	*destinataire*
`HPFC_SNDTO_C(what, val)`	*Computer*
`HPFC_SNDTO_H(what, val)`	`host`
`HPFC_SNDTO_A(what, val)`	tous les autres nœuds (canal *nodemulticast*)
`HPFC_HSNDTO_A(what, val)`	`host` pour tous les nœuds (canal *hostmulticast*)
`HPFC_SNDTO_O(what, val)`	*Owner*
`HPFC_SNDTO_OS(what, val)`	*Owners*
`HPFC_SNDTO_OOS(what, val)`	autres *Owners*
`HPFC_SNDTO_HA(what, val)`	tous les nœuds et `host` (canaux *nodemulticast* et `host`)
`HPFC_SNDTO_HNO(what, val)`	`host` et $n \notin$ *Owners*
`HPFC_SNDTO_N()`	*neighbour*, voisin courant

Enpaquetage des données pour un envoi, ou **dépaquetage** lors de la réception :

`subroutine`:

nom	*utilisation*
`HPFC_PACK_`n`(...)`	enpaquetage n dimensions
`HPFC_UNPACK_`n`(...)`	dépaquetage n dimensions

`arguments`:

argument	*signification*
`what, array`	nature du tableau et tableau de données
[lower, upper]n	bornes de chaque dimension du tableau
[low, up, inc]n	zone placée dans le paquet, pour chaque dimension

Réception de messages en provenance d'un des processus courant, selon les besoins de l'algorithme, et de manière à recevoir les messages envoyés avec les fonctions précédantes :

`subroutine` :

nom	*provenance*
`HPFC_RCVFR_S(what, val)`	*Sender*
`HPFC_RCVFR_H(what, val)`	`host`
`HPFC_RCVFR_C(what, val)`	*Computer*
`HPFC_RCVFR_mCS(what, val)`	*Sender* (canal *nodemulticast*)
`HPFC_RCVFR_mCH(what, val)`	`host` (canal *hostmulticast*)
`HPFC_RCVFR_N()`	*neighbour*, voisin courant

Paramètres des messages :

paramètre	*signification*
`what`	type de `val` pour PVM
`val`	valeur contenu du message

Calcul des nouveaux indices d'un tableau, après avoir vérifié que le processus courant est bien le propriétaire de la valeur en question, et détermination des bornes de boucles pour les itérations locales :

`integer function xx(arraynumber, dimension, indice)` :

nom	*fonction*
`HPFC_LOCALIND(...)`	calcul d'un indice local
`HPFC_LOCALINDGAMMA(...)`	calcul d'un indice local, cas γ
`HPFC_LOOP_BOUNDS(...)`	calcul des bornes de boucles locales

Paramètres des calculs d'indice :

paramètre	*signification*
`arraynumber`	numéro du tableau concerné
`dimension`	dimension du tableau
`indice`	valeur de l'indice initial

Paramètres du calcul des bornes de boucle :

paramètre	*signification*
`newlower, newupper,`	bornes de la boucle locale (calculée)
`init,`	borne globale initiale (calculée)
`lower, upper,`	bornes de la boucle globale
`arraynumber,`	numéro du tableau (*owner rule*)
`dimension`	dimension sur le tableau de processeurs

Initialisations des différentes descriptions pour chaque nœuds et pour `host`, appelées au moment du lancement :

`subroutine` :

nom	*fonction*
`HPFC_INIT_COMMON_PROCS`	initialisation des canaux de communication
`HPFC_INIT_COMMON_PARAM`	initialisation des descriptions, générée par le compilateur

Lancement du programme, par `host`, et auto-reconnaissance des nœuds pour connaître quel rôle HPF sera le leur, et sortie du programme :

`subroutine`:

nom	*fonction*
`HPFC_INIT_HOST`	lancement des nœuds ...
`HPFC_INIT_NODE`	initialisation des nœuds...
`HPFC_NODE_END`	sortie de PVM
`HPFC_HOST_END`	sortie de PVM

Autres fonctions non directement appelées mais utilisées par les précédantes:

`subroutine`:

nom	*fonction*
`HPFC_TEMPLATE`	calcul de l'élément de *template* sur lequel est aligné un élément de tableau
`HPFC_PROCESSORS`	calcul de l'élément de *processors* sur lequel est distribué un élément de *template*
`HPFC_PROCLID`	calcul du `lid` d'un élément de *processors*
`HPFC_CMPPOS`	calcul de l'eléments de *processors* correspondant à un `lid`

5.3.3 Utilisation de PVM 3.1.2

L'utilisation de PVM [GBD+93] dans sa version 3 est plus facile que dans la version precédente, car beaucoup des opérations qui nécessitaient des manipulations fastidieuses et répétitives ont été automatisées dans la nouvelle version, distribuée à partir de février 93.

5.3.3.1 PVM : *Parallel Virtual Machine*

PVM est une bibliothèque de fonctions qui permet d'échanger des messages (*message passing*) entre processus sur un réseau hétérogène. La librairie est disponible gratuitement en la demandant par courrier électronique, ou par `ftp`. Elle s'installe simplement sur un compte, et il suffit de compiler le programme pour les architectures souhaitées. Elle fonctionne actuellement dans la version 3.1.2 sur de nombreuses machines (voir figure 5.6). Les sources sont fournis, et l'installation ne pose pas de problèmes particuliers. Il est facile de faire quelques petits essais de programmes communiquants.

Il a été développé aux États-Unis par une équipe qui étudie les possibilités de résoudre de gros problèmes dans un environnement de machines hétérogènes. Les institutions impliquées sont ORNL (*Oak Ridge National Laboratory*), *Emory University*, *University of Tennessee*, *Carnegie Mellon University* et le *Pittsburgh Supercomputer Center*. Ce projet, commencé en 1989, est financé par divers organismes tels que *National Science Foundation*, *US Department of Energy*, et l'état du *Tennessee*.

PVM fournit un certain nombre de fonctions qui permettent de préparer des messages, en envoyer, en recevoir, ou encore faire des synchronisations, des rendez-vous, générer et utiliser des événements...On peut également demander la création de processus sur une certaine architecture, et un processus peut s'auto-identifier. L'ensemble de ces fonctions permet l'écriture de programmes répartis communiquants, se synchronisants, s'attendants...La librairie est très bien fournie, autant en ce qui concerne la diversité des machines acceptées que la documentation. Des pages de manuel dans le style `man` sont proposées.

PVM est écrit en C, mais une librairie permet d'utiliser ces fonctions à partir de Fortran. Les fonctionnalités sont les mêmes dans les deux langages. Il fonctionne sur station de travail en lançant un démon (`pvmd`) sur chaque machine qui doit être configurée en réseau hétérogène. Un processus qui fait une requête sur une machine fait appel aux services du démon qui prend les différentes actions en charge.

La sémantique des messages impliquée par le code produit suppose qu'il existe un canal virtuel entre les processus qui communiquent. L'ordre dans lequel les messages ont été envoyés doit être respecté à la réception. Ce fonctionnement doit être implanté en utilisant les fonctions fournies par PVM. Il est nécessaire pour assurer la justesse des communications, et donc la sémantique du programme. Cette implantation des canaux de communication est décrit en 5.3.3.2

5.3.3.2 Canaux de Communication

La sémantique des messages considérée par le modèle de compilation suppose l'existence d'une file de message entre chaque paire de processus. Elle suppose aussi que les processus sont capables de distinguer entre les envois en nombre et les messages particuliers envoyés par un même processus. Cette partie explique comment cette sémantique est garantie par l'utilisation judicieuse des numéros de message de PVM 3, qui n'oppose pas de telles contraintes dans son fonctionnement normal.

Les messages doivent être reçus dans l'ordre où ils ont été émis, et doivent être sélectionnés en fonction de leur provenance. PVM distingue les messages par un numéro spécifié au moment de l'envoi, et par leur provenance. Il faut donc que ce numéro permette de distinguer les messages en fonction de leur numéro d'ordre sur le canal virtuel de communication. En conséquence, les numéros d'ordre doivent être conservés par les envoyeurs et les receveurs. Des tables gardent donc trace des différentes communications qui ont eu lieu jusqu'à un point donné du programme, afin qu'un même numéro de message original soit créé en même temps par deux processus qui s'échange un message (envoi et réception).

Le cas des envois de un à tous (*multicast*) exige un traitement particulier. On emploie alors des canaux particuliers, selon que l'envoyeur est un nœud ou bien le processeur `host`. Afin d'éviter que puissent être confondus des messages venant des *envois en nombre*, et ceux allant simplement d'un processus à un autre, les numéros de messages sont dans le premier cas impairs et dans le second pairs.

Un résumé des canaux de communication employés entre les processus, selon les fonctions appelées, avec la première ligne pour l'envoyeur, et la seconde pour le ou les receveurs, est proposé à la figure 5.7.

5.4 Conclusion

Le schéma de compilation décrit au chapitre précédent a été implémenté au sein du projet Pips du Centre de Recherche en Informatique de l'École des mines de Paris. Le prototype de compilateur, appelé `hpfc` *(High Performance Fortran Compiler)*, est intégré dans le paralléliseur automatique Pips dont il profite, et profitera encore plus à l'avenir, des nombreuses analyses. L'implémentation du compilateur a été un développement important, qui a compris :

- un analyseur syntaxique pour les directives ;
- un module de normalisation du code ;

- la correction des déclarations pour les tableaux répartis ;
- l'implémentation de la double réécriture ;
- la programmation des fonctions de support pour la résolution dynamique ;
- le codage de l'analyse des recouvrements ;
- l'écriture de scripts en shell pour certaine partie de la chaine de compilation.

Le programme est appelé à l'aide des commandes de Pips, et produit 4 fichiers qui sont ensuite automatiquement compilés et linkés avec PVM, de manière à pouvoir exécuter les codes sur réseau de stations de travail. Le niveau du code généré était suffisant pour que soit lancé une campagne de mesures de performances présentée au chapitre suivant.

architecture	*machine*
ALPHA	DEC Alpha
CM2	Thinking Machine CM2
CM5	Thinking Machine CM5
CRAY	C-90, YMP, Cray-2
CRAYSMP	Cray S-MP
I860	Intel iPSC/860
KSR1	Kendall Square KSR-1
PGON	Intel Paragon
RS6K	IBM RS-6000
SUN4	Sun SparcStation

FIG. 5.6 - *Quelques unes des architectures fonctionnant sous PVM*

`receive from`

		C	S	H	mCS	mCH
	C	-	node node	-	-	-
	H	-	host node	-	-	-
	O	node node	-	node host	-	-
send to	Os	-	-	node host	-	-
	OOs	node node	node node	-	-	-
	HA	-	host node	-	mcastnode mcastnode	-
	HNO	-	host, node node	-	-	-
	A	-	-	-	mcastnode mcastnode	-
	H_A	-	-	-	-	mcasthost mcasthost

FIG. 5.7 - *Canaux virtuels utilisés selon les fonctions appelées*

Chapitre 6

Expérimentations

L'objet de cette section est de décrire les performances obtenues avec les codes générés sur un réseau de stations de travail configurées sous PVM 3, en présentant tant les performances absolues (temps d'exécution, Mflops) que les performances relatives (accélérations mesurées). Ces performances sont ensuite comparées avec celles obtenues avec d'autres compilateurs, tels Adaptor [Bra93b, Bra93a, Bra92d, Bra92c, Bra92a, Bra92b] ou Fortran D [HKT92a, HKT92b, MFvL+92, Tse93].

6.1 Performances

Les performances ont été mesurées sur un réseau de stations de travail reliées par *Ethernet*, pour une application très classique, une relaxation Jacobienne.

6.1.1 Compilateurs et Options de Compilation

Deux langages ont été directement ou indirectement utilisés pour compiler le code :

C : le compilateur `gcc` a été utilisé avec les options `-Wall` (vérifications diverses du code) et `-O` (optimisations), pour compiler PVM 3.

Fortran 77 : le compilateur `f77` de Sun a été utilisé, avec l'option `-O4` (optimisations maximales), pour compiler le code produit et la librairie de fonctions de support.

6.1.2 Application

L'application testée (figure 6.1) est une relaxation Jacobienne de 100 itérations sur des matrices $n \times n$ de flottants en simple précision. Par soucis de réalisme, quelques entrées-sorties ont été ajoutées, ce qui permet de vérifier la concordance des résultats entre exécutions séquentielle et parallèle. Pour une matrice de taille n, le nombre de calculs en virgule flottante effectués est de $400(n-2)^2$ (avec 1 multiplication pour 3 additions).

Cette application a été choisie parce qu'elle permet de faire des comparaisons avec d'autres travaux disponibles dans la littérature [Bra92d, HKT92b]. Elle obtient de très bonnes performances, car le volume des communications nécessaires progresse linéairement avec la taille de la matrice n alors que les calculs sont quadratiques (n^2).

Les tests ont été faits pour différentes tailles de matrices, en comparant les performances des versions séquentielle et parallèle pour chaque taille, tant en absolu (mesure du temps d'exécution total, des performances en calcul flottant) qu'en relatif (mesure de l'accélération

```
      program hpftest41
      real tc(500,500), ts(500,500), north(500)
chpf$ template t(500,500)
chpf$ align tc(i,j), ts(i,j) with t(i,j)
chpf$ align north(i) with t(1,i)
chpf$ processors p(2,2)
chpf$ distribute t(block,block) onto p
      print *, 'hpftest41 running'
      print *, 'thermo'
c
c initialization
      print *, 'initializing'
c initializations of north, tc and ts...
c ...
c iterations...
      print *, 'running'
      do k=1,100
c
c computation and copy back (dataparallel semantic)
chpf$ independent(j,i)
         do j=2,499
            do i=2,499
               ts(i,j) = 0.25*
     $                 (tc(i-1,j) + tc(i+1,j) + tc(i,j-1) + tc(i,j+1))
            enddo
         enddo
chpf$ independent(j,i)
         do j=2,499
            do i=2,499
               tc(i,j) = ts(i,j)
            enddo
         enddo
      enddo
c
c print results
      print *, 'results:'
 10   format(F8.2, F8.2, F8.2, F8.2, F8.2)
      do i=2, 10, 2
         write (6,10) tc(i,2), tc(i,4), tc(i,6), tc(i,8), tc(i,10)
      enddo
      print *, 'hpftest41 ended'
      end
```

FIG. 6.1 - *Programme 41, Relaxation Jacobienne, $n = 500$*

nom	type	Mflops
`calloway`	SS10	5.27
`moret`	SS2	3.00
`gramat`	ELC	2.36
`nemours`	SS1+	1.79
`dvorak`	SS1	1.45

TAB. 6.1 - *Performances de quelques machines sur une même application*

obtenue). Il est important de noter que le nombre d'itérations effectuées est ridiculement faible par rapport à ce qui est nécessaire pour espérer voir converger l'algorithme sur un exemple réel. Les mesures effectuées ne donnent donc pas une idée de l'extensibilité de l'algorithme de relaxation, mais uniquement des performances qui peuvent être atteintes par le code généré (on peut se rapporter pour une analyse fine des problèmes d'extensibilité à l'article de Singh, Hennessy et Gupta [PSHG93]).

Dans la version parallèle de l'application, les matrices sont distribuées par *block* sur les deux dimensions, et les processeurs sont arrangés en grille.

6.1.3 Matériel

Le tableau 6.1 présente les performances relatives de différentes machines de la gamme de stations de travail Sun pour une même application, une relaxation Jacobienne de 100 itérations sur matrice 500×500. Ces chiffres seront pris comme performance nominale de ces machines pour ce type d'application dans les différentes formules proposées dans les tableaux et figures qui suivent.

Les tests multiprocesseurs ont été faits sur des Sparc Station 1 (SUN 4/60), car c'est le seul type de machine dont on disposait en assez grand nombre pour faire des tests sur réseau de stations homogènes. La relative lenteur de ces machines amméliore de manière sensible les résultats obtenus, dans la mesure ou la lenteur des calculs permet de ne pas mettre trop en évidence la faiblesse du réseau d'interconnexion.

Ces stations sont reliées par un réseau *Ethernet*, dont on supposera le débit pratique au niveau *socket* utilisé par PVM de 700 Ko/s ($1.4\mu s$/octets) et la latence par message de 1.1 ms, en s'appuyant sur les nombreuses mesures faites par Éric Bazin dans [Baz91]. Le système de communication entre stations de travail est tel que les messages (paquets) sont séquentialisés au niveau du réseau, ce qui est un très pénalisant par rapport à une machine parallèle qui peut échanger des messages entre processeurs différents en parallèle.

6.1.4 Mesures des Performances

Les temps d'exécution ont été mesurés à l'aide de la fonction `/usr/bin/time`, en prenant en compte le temps global d'exécution. Les résultats retenus sont les meilleurs temps sur un série d'au moins 5 exécutions, voir plus pour certaines mesures douteuses. On espère ainsi éviter les *artefacts* dûs au système d'exploitation et les éventuelles interférences avec d'autres usagers. Les variations des temps d'exécution entre deux essais est parfois importante, et ce d'autant plus que le nombre de machines impliquées est grand. Ces mesures ont eu lieu sur un réseau peu chargé, pendant la période estivale. Les graphiques proposés ont toujours en abscisse la taille du problème n.

Les résultats sont présentés de différentes manières, en utilisant plusieurs indicateurs.

Les temps d'exécution sont d'abords donnés, tels qu'ils ont été mesurés. On propose ensuite le nombre de Mflops déduit de cette mesure, et connaissant le nombre d'opérations en virgule flottante de l'application. Pour les tests en parallèle, ce même graphique est présenté sous la forme d'une efficacité, c'est à dire le pourcentage atteint par l'application par rapport à la performance de crête obtenue en séquentiel avec le code initial.

Enfin pour les tailles de matrice pour lesquelles on dispose à la fois d'une mesure de temps séquentiel et parallèle, on présente l'accélération (*speedup*), c'est à dire le rapport entre les temps séquentiel et parallèle. Il faut manier ce dernier indicateur avec prudence, car de très bonnes accélérations peuvent être atteinte pour certaines valeurs de taille de matrice, si l'exécution en séquentiel ne se passe pas bien, généralement à cause de phénomènes de cache.

6.1.5 Exécutions Séquentielles

Une première série de tests a été faite pour différentes valeurs de n, et deux types de parcours des matrices, pour mesurer les éventuels effets de cache : le programme `40` parcourt les matrices en commençant par la seconde dimension, alors que le `41` commence par la première, ce qui doit lui permettre de profiter d'effets de cache puisque les données sont lues et écrites de manière contiguë en mémoire. Les résultats sont présentés figures 6.2 et 6.3.

La figure 6.2 présente les meilleurs temps d'exécution mesurés. La courbe théorique proposée traduit les temps d'exécution si les performances sont de 1.45 Mflops quelle que soit la taille du problème considéré. On remarque une bonne corrélation avec les points de mesure, plus quelques incidents favorables ou défavorables. Les incidents défavorables peuvent être expliqués, du moins pour certain, par des problèmes de cache, dont une discussion détaillée n'est pas justifiée ici. Les incidents favorables sont également liés à des problèmes de cache, puisque les valeurs pour lesquelles ces incidents se produisent sont très proche de la taille limite permettant aux 2 matrices du problème de tenir entièrement dans la mémoire cache[1].

La figure 6.3 présentes les performances obtenues en calcul flottant, déduites des mesures en temps. On constate qu'à de rares exceptions près, le programme `41` se comporte mieux que le programme `40`, et qu'il y a plus d'accident pour cette dernière version de la relaxation. Autour de la taille $n = 100$ de nombreuses mesures ont été effectuées, en raison de la présence de très bonnes performances dans ce voisinage lors des premiers essais. Au grain le plus fin d'observation, l'irrégularité des performances observées est indéniable, mais pas surprenante compte-tenu d'effets de cache pour ces tailles de matrice et ce type de machine.

6.1.6 Exécution en parallèle sur 4 Processeurs

Une première série de tests avec le code produit par le compilateur a été faite avec 4 processeurs. Les figures 6.4, 6.5, 6.6 et 6.7 présentent les divers résultats obtenus.

La figure 6.4 donne les temps d'exécution réels de l'application en parallèle sur 4 processeurs pour différentes valeurs de la taille de la matrice. La courbe « théorique » est basée sur une extrapolation des performances avec ce type de processeur sur cette application à partir des performances de l'application séquentielle d'origine, sur l'évaluation du temps de transfert pour les différents messages nécessaires, proportionnel à la taille de la matrice, et sur une approximation à vue de la latence globale, liée au lancement du programme avec PVM, à la latence des opérations de communication et aux entrées-sorties dont le volume est fixé.

1. 64 Ko sur les SUN 4/60

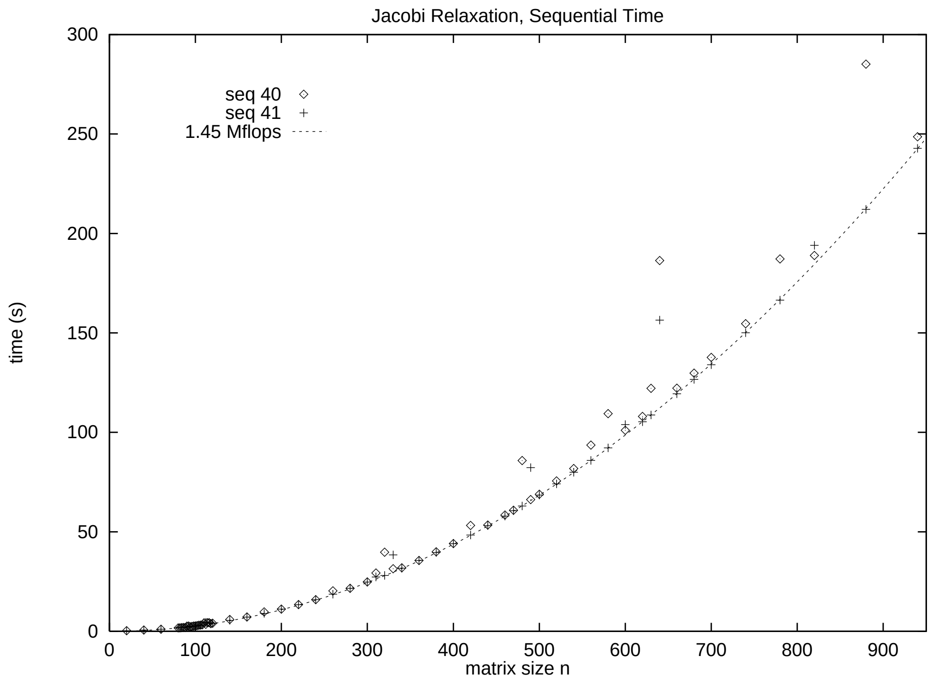

FIG. 6.2 - *Temps, exécution séquentielle*

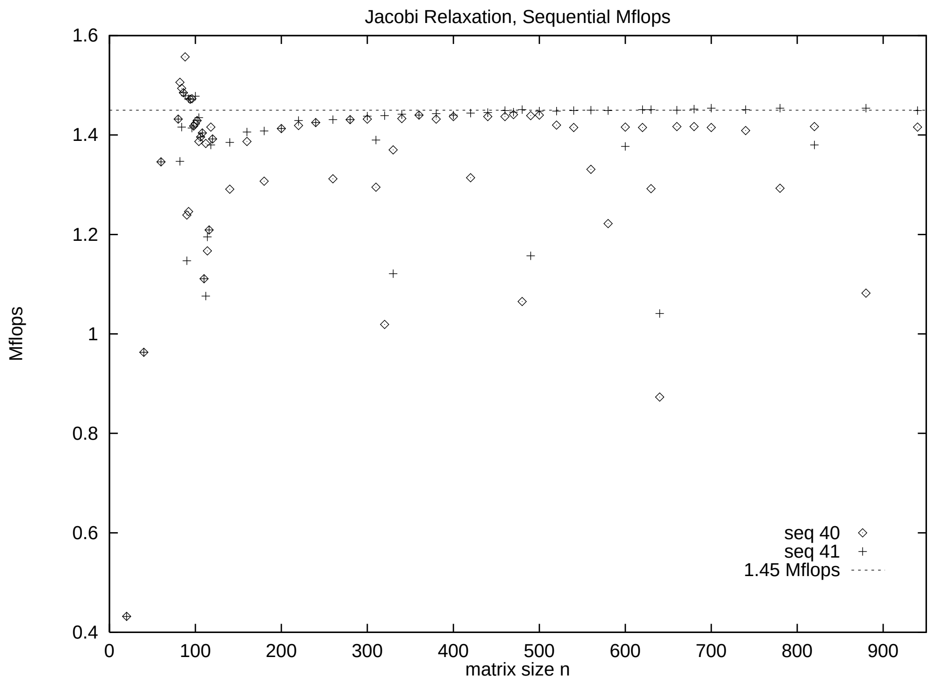

FIG. 6.3 - *Performances en calculs flottants*

Le temps ainsi estimé est le temps nécessaire à une application pour effectuer les calculs et communications de la relaxation, sauf à envisager la possibilité de recouvrement entre calculs et communications. On pourrait alors espérer gagner les temps de communications sur l'application entière, ce qui permettrait de relever les performances pour les exécutions avec n petit.

La figure 6.5 donne les performances en calculs flottants déduites des mesures de temps d'exécution. On constate que, plus clairement encore que dans la version séquentielle, les boucles favorisants le parcours sur la première dimension se passent bien. De plus, les points expérimentaux suivent le même comportement que la courbe « théorique », à quelques incidents près.

La figure 6.6 présente les mêmes résultats, mais en les comparant à la puissance de crête des 4 processeurs pour cette application. On constate que pour des tailles suffisamment importantes, l'efficacité dépasse les 80%. Par contre, des tailles trop faibles provoquent un effondrement, la lenteur des communications écrasant tout le gain espéré avec l'exécution en parallèle. Il est cependant bon de noter que des tailles de problème de l'ordre du millier sont réalistes [Baz91, HKT92b, Ani93], et leur parallélisation présente réellement un intérêt pratique.

Enfin la figure 6.7 décrit l'accélération obtenue. Les très bons résultats obtenus correspondent non pas à des très bonne exécutions des versions parallèles, mais plutôt à des exécutions mauvaises de la version séquentielle. Les « bonnes » accélérations tout comme les mauvaises sont à interpréter avec méfiance et attention, en regardant d'éventuelles chutes de performance des versions séquentielle et parallèle. L'indicateur d'accélération n'est, à l'évidence, pas un très bon paramètre.

Les résultats montrent que le compilateur `hpfc` permet d'atteindre de bonnes performances, proches des performances théoriques qui peuvent être espérées sur ce type de machine et ce type de réseau d'interconnexion, pour une application qui, il est vrai, se prêtent particulièrement bien aux optimisations qui ont été implémentées dans le propotype de compilateur `hpfc` (voir section 5.2.5).

6.1.7 Exécution en parallèle sur 8 Processeurs

Une seconde série de tests a été faite avec cette fois 8 processeurs. Il s'agit de la version `42` du programme. Les tailles de matrice explorées ont été jusqu'à 2000, mais aucune accélération n'est proposée pour les tailles au delà de 1000 : pour une exécution en séquentiel sur de telles données, la mémoire physique des machines est insuffisante, et le mécanisme de mémoire virtuelle va dégrader de manière significative les performances. Comme de toute façon l'accélération n'est pas un très bon indicateur, ça ne dérangera personne. Les résultats de ces mesures sont présentés aux figures 6.8, 6.9, 6.10 et 6.11.

La figure 6.8 présente les temps d'exécution mesurés pour la relaxation Jacobienne sur 8 processeurs. On constate que les mesures se placent *grosso modo* sur la courbe théorique, à quelques « incidents » près.

Les figures 6.9 et 6.10 montrent les performances en calculs flottants et l'efficacité qui peut être déduite de ces mesures. On retrouve les divers « incidents » qui sont plus clairement mis en valeur. Pour de grandes tailles de la matrice, on peut remarquer que les valeurs mesurées sont quelques pour cent au dessus des valeurs théoriques. Par contre, pour $n = 1400$ et $n = 1600$, la valeur mesurée est significativement en dessous, malgré de nombreuses mesures. On peut supposer que des problèmes de caches sont à l'origine de ces variations, mais il est difficile d'en être sûr.

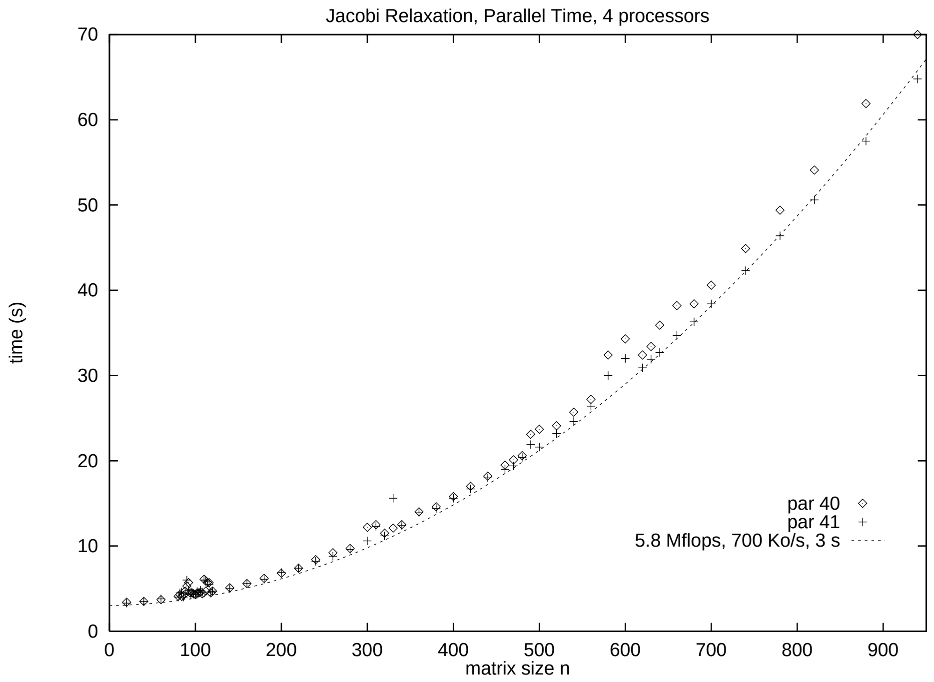

FIG. 6.4 - *Temps en parallèle, 4 processeurs*

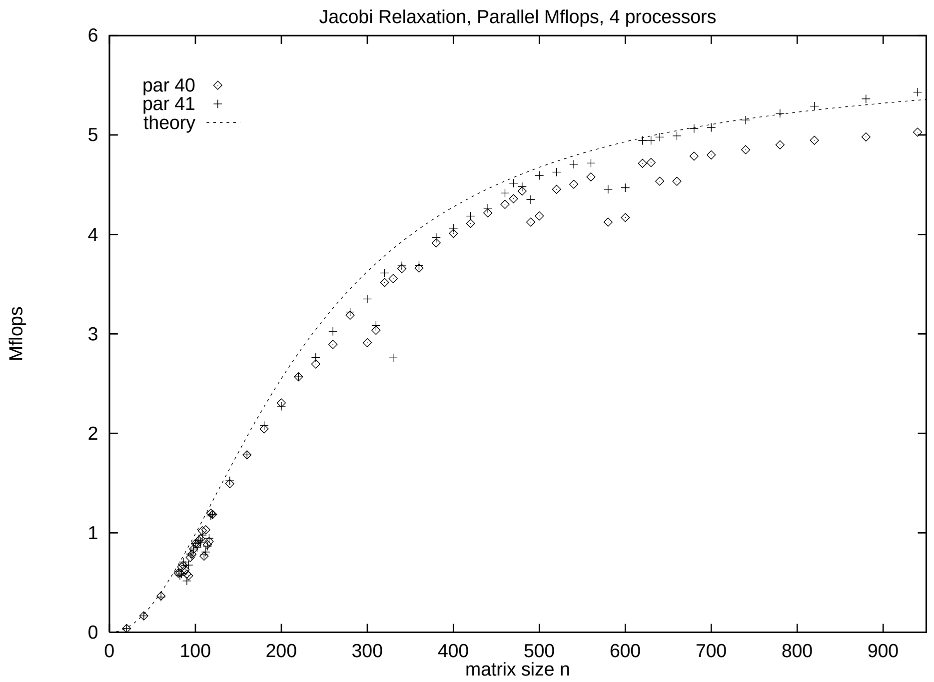

FIG. 6.5 - *Performances en calculs flottants, 4 processeurs*

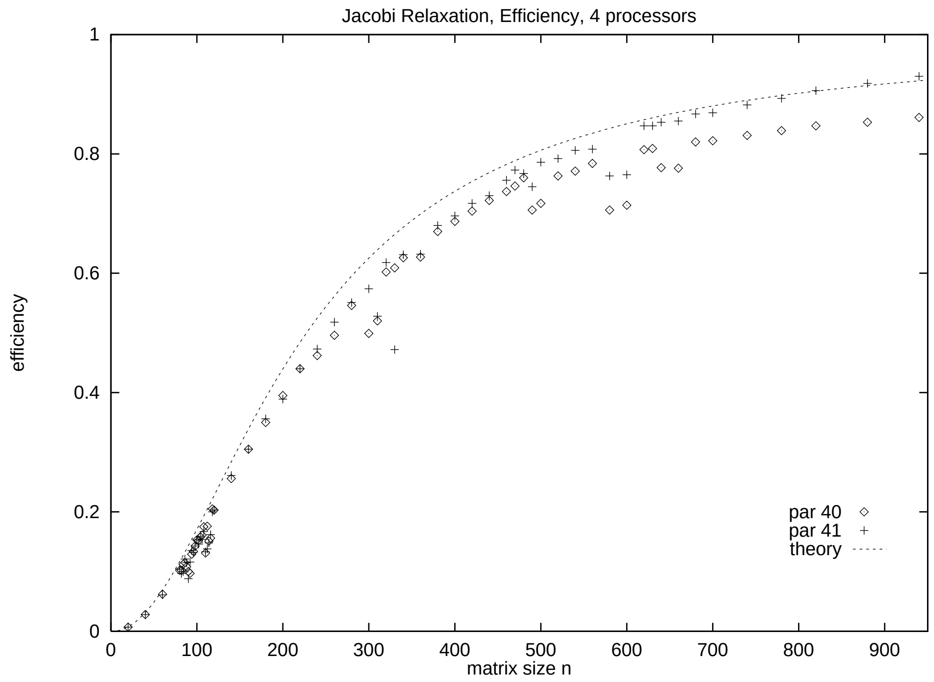

FIG. 6.6 - *Efficacité, 4 processeurs*

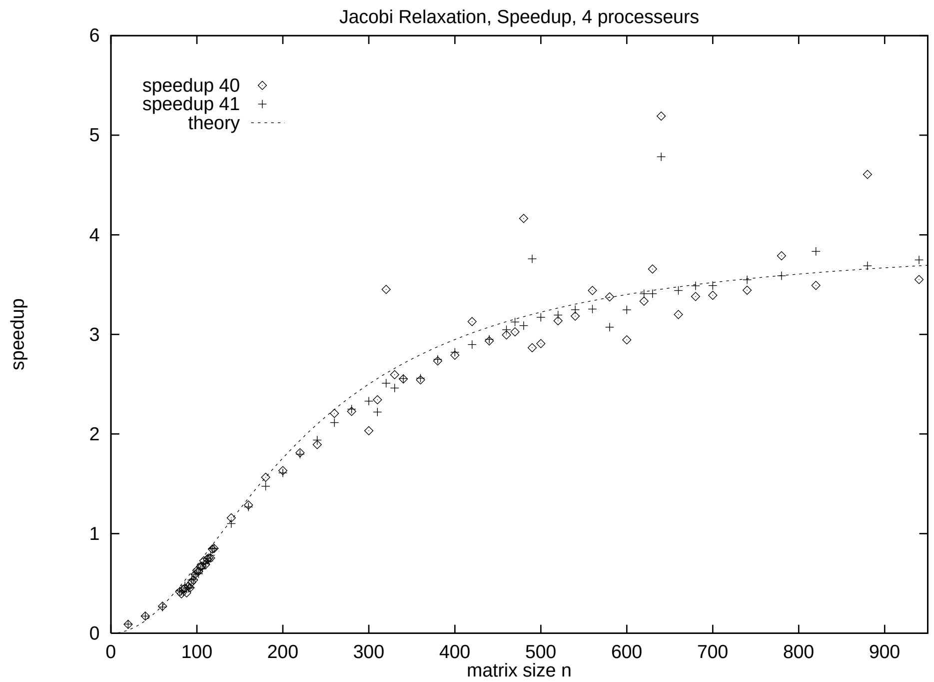

FIG. 6.7 - *Accélération, 4 processeurs*

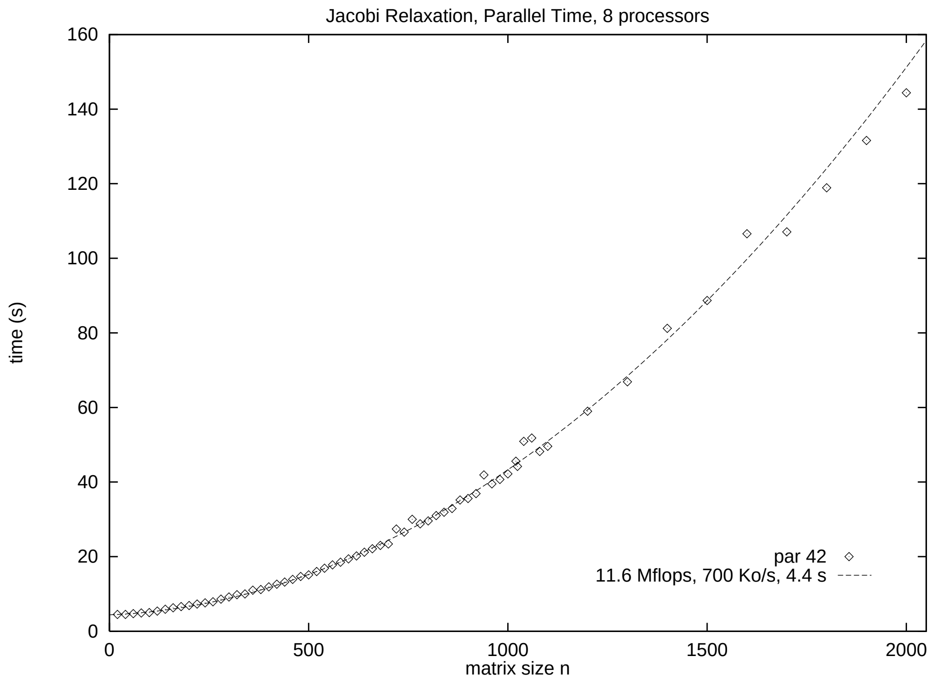

FIG. 6.8 - *Temps en parallèle, 8 processeurs*

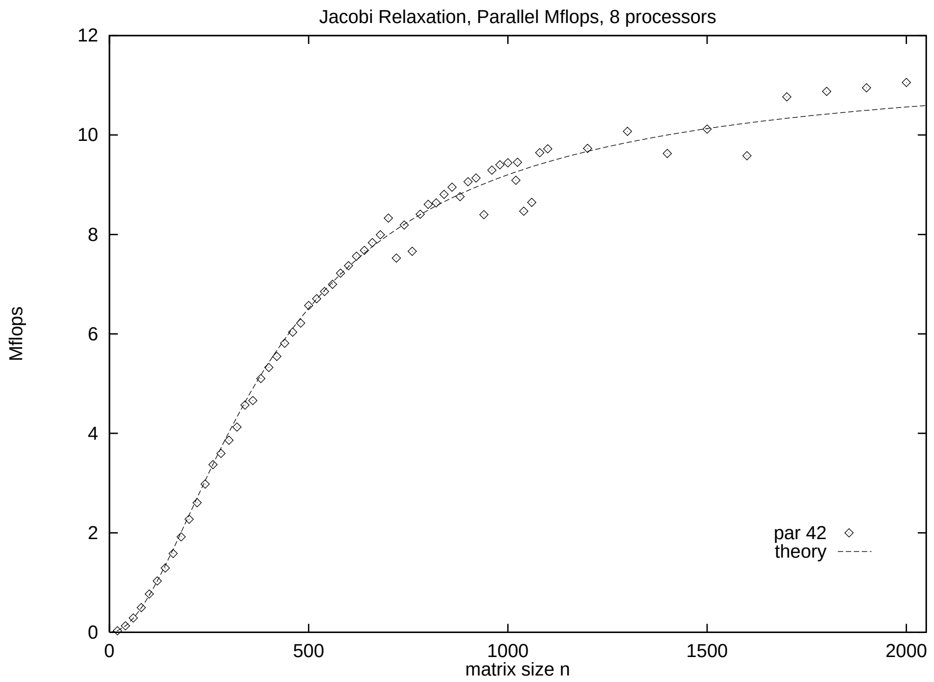

FIG. 6.9 - *Performances en calculs flottants, 8 processeurs*

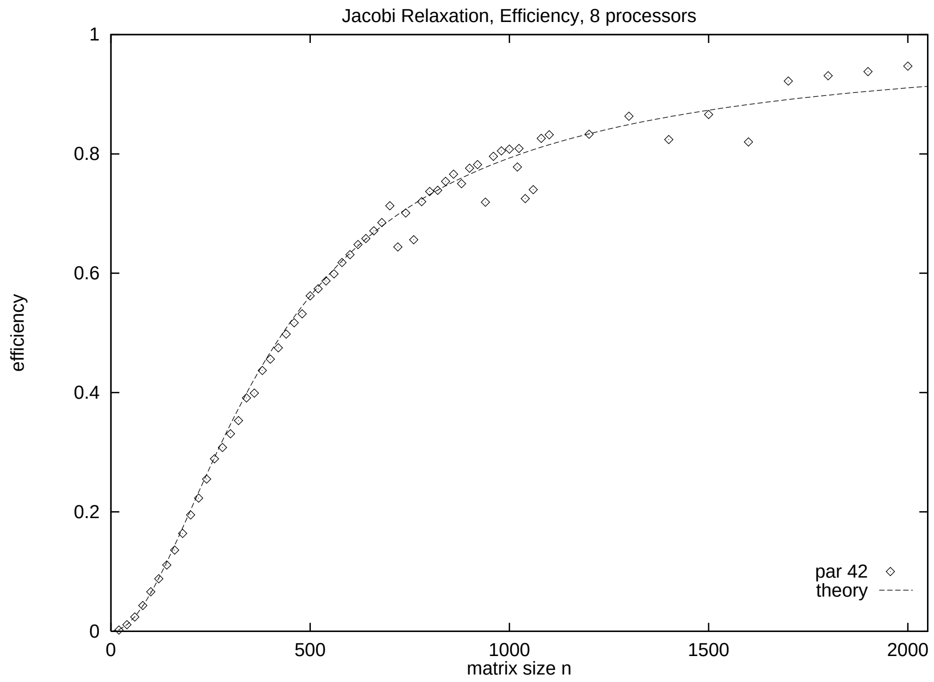

FIG. 6.10 - *Efficacité, 8 processeurs*

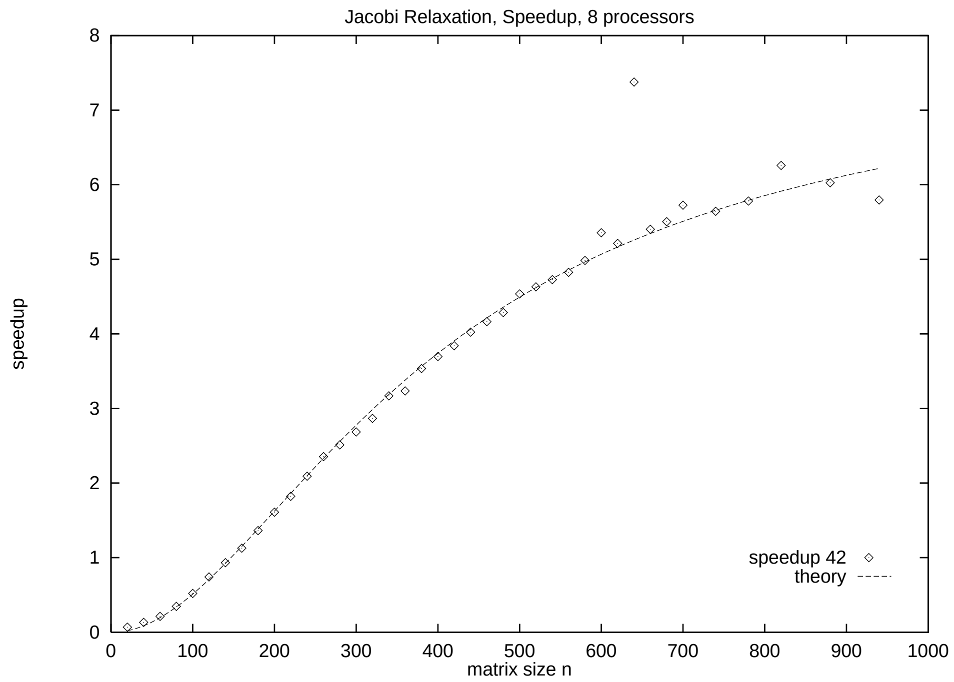

FIG. 6.11 - *Accélération, 8 processeurs*

Enfin la figure 6.11 présente l'accélération obtenue lorsque le temps séquentiel est connu, et avec les mêmes réserves que dans l'analyse précédente.

6.1.8 Appréciation des Performances

Pour ce type d'application, où tout se passe bien, le compilateur `hpfc` génère un code au niveau des performances optimales qui peuvent être espérées compte tenu des processeurs et du réseau d'interconnexion les reliant, comme le montre la correspondance entre les meilleures mesures effectuées et les courbes « théoriques » proposées, qui s'appuient sur les performances nominales des processeurs et du réseau de communication. Si l'on admet la possibilités de recouvrement des calculs et communications, la courbe théorique de l'efficacité monterait beaucoup plus vite vers la valeur de crête. On peut donc espérer de meilleures performances encore avec cette optimisation supplémentaire.

On ne doit pas oublier les particularités de cette application qui permet d'atteindre facilement de bonnes performances avec une distribution automatique, et les nombreuses conditions qui sont nécessaires au bon déroulement des algorithmes d'optimisation (voir section 4.3.1). Pour des applications plus générales, de telles performances seront plus difficiles à atteindre. On a cependant montré que le code généré était aussi bon que celui qui pourrait être écrit manuellement, puisque les ressources sont saturées, sauf à envisager des recouvrements entre calculs et communications.

La figure 6.12 compare de manière théorique les performances qui peuvent être obtenus en superposant ou non les calculs et les communications. Si les calculs recouvrent les communications, l'efficacité se rapproche plus vite de l'optimum.

6.2 Comparaisons

Divers systèmes de compilation sont disponibles ou en cours de développement, et des essais ont été faits avec les codes générés. On cherche ici à comparer ces outils avec notre prototype de compilateur, en ce qui concerne tant les performances obtenues que la généralité des algorithmes implémentés.

6.2.1 Adaptor

Le système de compilation Adaptor a été développé par Thomas Brandes [Bra93b, Bra93a, Bra92d, Bra92c, Bra92a, Bra92b] au *High Performance Computing Center, German National Research Institute for Computer Science (GMD).* Ce système prend en entrée du Fortran 77 avec quelques traits de Fortran 90 (sections de tableau...) et des directives de placement limitées (distribution de la dernière dimension des tableaux de données uniquement), pour produire du Fortran 77 avec échange de messages, en utilisant la librairie PVM ou d'autres librairies dédiées à certaines machines. Adaptor est un outil « pragmatique », qui permet de compiler des codes simples de manière efficace.

Le tableau 6.2 reproduit les informations disponibles concernant une relaxation Jacobienne compilée avec Adaptor. La taille du problème considéré est fixée. Elle est très faible par rapport à ce qui est nécessaire pour obtenir des performances raisonnables sur un réseau de stations de travail. De plus, la machine sur laquelle ont été faits les essais est une machine à mémoire partagée, d'architecture et de fonctionnement assez différents d'une machine à mémoire répartie. On n'a pas de point de comparaison (à travers une modélisation, ou un encodage à la main de l'algorithme) pour évaluer les performances atteintes avec ce que

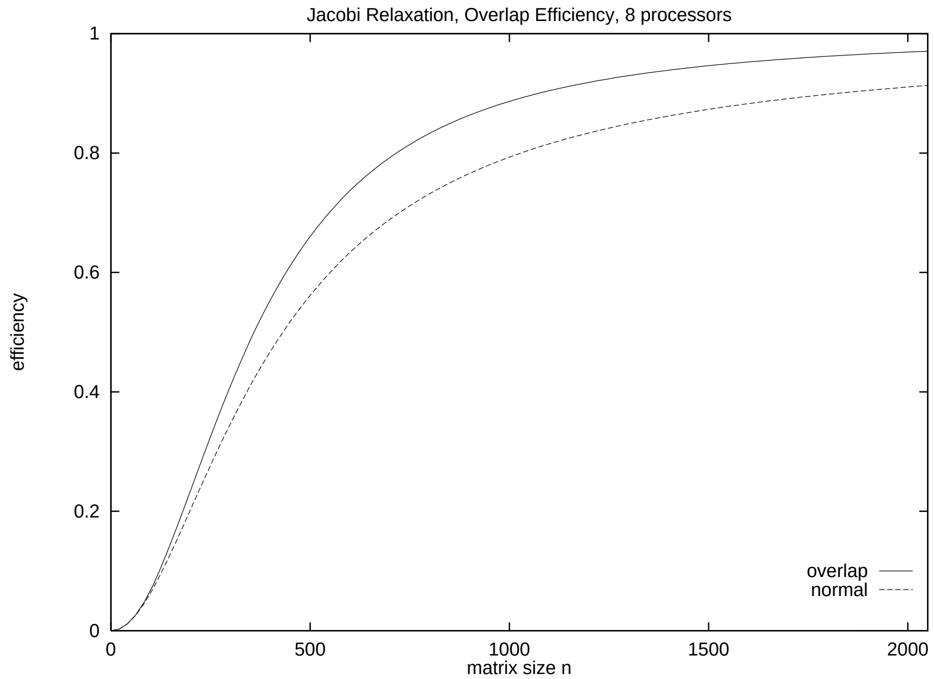

FIG. 6.12 - *Avec et sans Recouvrements, 8 processeurs*

programme	temps (s)	accélération	efficacité (%)
original	93.4	1.0	100
1 nœud	102.6	0.9	91
2 nœuds	56.1	1.7	83
4 nœuds	30.2	3.1	77
8 nœuds	18.5	5.0	63

TAB. 6.2 - *Relaxation Jacobienne* 128×128, *Adaptor, sur Alliant FX/2816*

taille	temps seq.	temps par.	accélération
128	20.8	4.1	5.07
256	84.8	9.8	8.65
512	331.2	28.3	11.70
1K	1550.4	110	14.10
2K	6160	412	14.95

TAB. 6.3 - *Relaxation Jacobienne, Fortran D, sur Intel ISPC/860, 16 proc.*

permettrait la machine. L'auteur indique juste dans [Bra92d] que c'est mieux que la parallélisation (et le placement) automatique faits par le compilateur Fortran fournit par le constructeur. Pour ce qui est des entrées-sorties, il n'est pas précisé si l'application présentée en comportait ou non.

Il faut noter que de très nombreux résultats sont disponibles pour Adaptor, qui a été testé sur le *Purdue Set*, un ensemble de programmes représentants les divers types d'applications qui peuvent être exécutées sur des machines à mémoire répartie. Ces résultats sont présentés dans [Bra93b, Bra93a]. Les applications testées sont de vraies applications (avec entrées-sorties, etc.) qui ont été un peu adaptées de manière à pouvoir être compilées par le sous-ensemble de Fortran autorisé par Adaptor.

6.2.2 Fortran D

Fortran D est l'une des extensions de Fortran pour machines parallèles à mémoire répartie qui a le plus influencée la spécification du *High Performance Fortran*. Le langage a été défini à l'Université de Rice par l'équipe de Ken Kennedy [HKT92a, HKT92b, MFvL+92, Tse93]. Le prototype de compilateur disponible a été implémenté par Chau-Wen Tseng au cours de son travail de thèse [Tse93]. Quelques mesures de performances sont exposées au chapitre 6 de sa thèse, qui reprend l'article [HKT92b].

Il est à noter que sur les mesures effectuées, seul un nombre limité sont liées à des codes généré automatiquement par le prototype de compilateur. L'objet de l'article est plus de savoir quelles optimisations sont vraiment intéressantes pour obtenir les meilleures performances. L'une des applications discutée est justement une relaxation Jacobienne, et les résultats sont présentés au tableau 6.3, en ne retenant cependant que les codes générés automatiquement.

De bonnes performances sont obtenues pour des matrices, même de faible taille. Pour les matrices de taille importante, l'accélération est excellente... avec toutes les réserves déjà exprimées sur cet indicateur trompeur. Il faut aussi remarquer que le temps séquentiel proposé n'est pas mesuré, mais extrapolé à partir de l'exécution du programme des nœuds auquel les communications ont été retirées. Ce procédé est très discutable, car ce code peut être très éloigné de la version séquentielle réelle. Ceci est justifié parce que la mémoire des nœuds n'est pas assez grande pour exécuter le code séquentielle, mais c'est faux pour une bonne partie des valeurs proposées.

Une analyse fine des temps d'exécution séquentiels donnés montre que les performances de l'application se dégradent d'environ 16% pour les deux dernières valeurs de taille de matrice, ce qui ammène à douter de la valeur de l'accélération fièrement annoncée en conclusion du tableau. En effet, en supposant le temps de calcul quadratique $t = an^2$, on constate que le paramètre a devrait avoir pour valeur 0.00126 pour les trois premières tailles de matrice et 0.00146 pour les deux dernières.

Pour ce qui est de la compilation, les algorithmes implémentés dans le prototype ont été très simplifiés. Ainsi, le compilateur n'accepte que des distributions sur une seule dimension des tableaux, ce qui simplifie considérablement la génération des messages qui doivent être échangés entre voisins. Le compilateur Fortran D n'accepte pas d'entrées-sorties, ce qui ne permet donc pas de vérifier la justesse des opérations faites.

Une seconde série de tests est décrite dans la thèse de Tseng, au chapitre 9. Pour la relaxation Jacobienne, des mesures ont été faites avec une CM-5, et le compilateur Fortran D est comparé au compilateur de CM-Fortran proposé par Connexion Machine.

6.2.3 Conclusion

Les résultats obtenus sont encourageants, et le prototype de compilateur `hpfc` est comparable avec d'autres projets. Les optimisations implémentées sont il est vrai moins développées, mais ces résultats ont été obtenus rapidement. Il reste encore à effectuer des expériences sur des codes de nature différente.

Chapitre 7

Conclusion

Un certain nombre de remarques sont nécessaires pour expliciter les problèmes actuels du compilateur d'une part, mais aussi pour mettre en valeur les solutions à venir et les résultats déjà obtenus. Une discussion est proposée sur les problèmes inhérents aux optimisations pour un algorithme de résolution dynamique des accès aux variables réparties tel que $\mathcal{T}_0$.

7.1 Problèmes rencontrés

Deux problèmes ont été principalement rencontrés lors du développement du compilateurs, qui sont liés à l'environnement de programmation utilisé, à savoir celui du paralléliseur automatique Pips. Le premier problème est lié au calcul des effets de Pips, c'est à dire la manière dont il analyse le fait que des variables soient touchées par les instructions du programme, en lecture ou écriture. Le second problème dépend des informations de typage présente dans Pips, pour traiter les expressions.

7.1.1 Effets

Les effets calculés par Pips ne correspondent pas exactement à ce dont le compilateur a besoin, et ils ne sont pas utilisés. Les problèmes liés au calcul des effet, illustrés par l'exemple proposé à la figure 7.1 sont les suivants :

```
program effects
integer i,j,k
i = 1
j = 2
k = 3
call compute(i,j,k)
call compute(i,j,j)
call compute(i,i,i)
end
subroutine compute(i,j,k)
i = i+j
end
```

FIG. 7.1 - *Exemple pour le calcul des effets*

- les paramètres des fonctions qui ne sont pas touchées sont retirés des effets, or on a quand même besoin de les remplacer dans la phase de réécriture de l'algorithme de compilation ;

```
C               <MUST BE READ   >: J I
C               <MUST BE WRITTEN>: I
      CALL COMPUTE(I, J, K)
```

- les effets multiples sur certaines variables sont fusionnés, mais on a alors pas accès aux références concernées dans l'appel de fonction, ce qui oblige à aller à la pêche ;

```
C               <MUST BE READ   >: J I
C               <MUST BE WRITTEN>: I
      CALL COMPUTE(I, J, J)
```

- les effets en lecture ne gardent que la référence, mais il est difficile éventuellement de savoir si la même référence est lue ;

```
C               <MUST BE READ   >: I
C               <MUST BE WRITTEN>: I
      CALL COMPUTE(I, I, I)
```

- enfin on peut discuter du niveau auquel doit être tenu une référence pour la remplacer sans problème : on peut soit tenir la référence directement (`reference`), soit à un niveau au dessus, en attrapant le `syntax` qui contient la référence.

Les effets qui devront être développés par la suite pour être exploités par le compilateur de bonne manière devront, pour chaque référence d'un appel à une fonction avec effets de bords, donner l'information de lecture et/ou écriture ou rien sur cette référence, ainsi que donner la liste des variables indirectement modifiées par des effets de bords. Ce n'est pas encore le cas, mais l'adaptation du calcul des effets de Pips à cette fin ne devrait pas poser de problèmes insurmontables.

7.1.2 Typage

Le second problème rencontré dans le développement du compilateur par rapport à Pips est la faiblesse du traitement du typage par le compilateur, qui ne type rien au dessus des variables. Dans le cas des expressions invoquant le moindre intrinsèque, la seule information disponible est donc *overloaded*, ce qui est un peu juste pour déclarer une variable temporaire devant recevoir le résultat de l'évaluation de l'expression. Par défaut, en cas de doute, le type déclaré de la variable est entier, et un avertissement est affiché pour l'utilisateur.

7.2 Liens avec les autres optimisations

Le code généré par le *High Performance Fortran Compiler* peut bénéficier grandement d'optimisations en amont, pendant la compilation et en aval de celle-ci sur le code produit, et il est intéressant de se poser un certain nombre de questions à ce sujet. Quelques mots sur les quelques optimisations déjà faites par le compilateur figurent en 5.2.5.2.

***Dead code* généré :** l'algorithme de compilation génére parfois du code mort, c'est à dire du code qui ne sert visiblement à rien, par exemple pour définir une variable qui n'est pas utilisée pas la suite. Ces codes doivent être retiré du programme puisqu'ils représentent un somme de calculs inutiles et donc une perte de temps.

L'originalité du code généré est que l'on peut y trouver du code mort, pour définir des variables inutilisées par exemple, mais dont les fonctions de définition, de communication par exemple, ont des effets de bords (la réception du message et l'incrémentation du compteur du nombre de message sur un canal virtuel).

Le retrait de ce code mort doit donc être fait en corrigeant parallèlement les communications entre les nœuds et `host`, ou entre nœuds, pour que tout se passe bien et que les messages ne soient pas envoyés. On doit donc garder le lien entre les envois et réceptions de messages complémentaires, afin d'éliminer le code en toute correction. De plus, il faut prendre garde à corriger les graphes utilisés (*use-def chains* par exemple) en conséquence, afin de faire éventuellement remonter l'élimination par fermeture transitive à partir des feuilles inutiles.

Cette élimination de code mort ne peut pas être fait par le compilateur Fortran utilisé en aval de HPFC, car elle implique une intervention en parallèle sur les codes produits, `host` et `node`, en exploitant en même temps les graphes de flot de données de ces deux modules.

Propagation et Évaluations : Le code compilé peut profiter à de multiples niveaux d'une phase de propagation de constantes et d'évaluation partielle quand c'est possible. La remontée des invariants de boucles est également à mettre dans cette partie. On a donc :

- en amont, la propagation de constante peut permettre de détecter qu'une certaine forme d'optimisation est faisable, ce qui n'aurait pas été le cas autrement ;
- en aval, le code généré dans certains exemples pourrait profiter d'une évaluation partielle de certains calculs, comme le calcul des nouveaux indices d'une variable répartie quand on connait statiquement l'indice global auquel l'accès est fait.

Détection des sous-expressions communes : Le code généré introduit de nombreuses variables égales les unes aux autres, et il est intéressant d'éliminer cette redondance inutile en les fusionnant, ou en fusionnant leur parties de calcul en commun.

Mémoire Enfin les différents processus de la compilation, comme la normalisation du code à compiler ou bien les réécritures, génèrent de très nombreuses variables nouvelles à courte durée de vie, qui pourraient avantageusement être fusionnées afin d'éviter l'allocation d'espace mémoire inutile. Certaines variables generées sont même totalement inutiles (ce qui doit pouvoir être corrigé d'ailleurs).

7.3 Continuation

Ce projet de compilateur en est à la version préliminaire du prototype qui doit servir de base à l'élaboration d'un compilateur non seulement plus complet, mais également optimisant. Il est donc normal d'envisager sa continuation et les pistes à suivre pour améliorer son fonctionnement et étendre le langage accepté.

7.3.1 Développement

Le développement du compilateur doit être poursuivi à divers points de vue :

- corriger les bogues répertoriés ;
- programmer un calcul d'effet comme souhaité, et faire les modifications conséquentes sur le compilateur ;
- vérifier les conditions nécessaires ;
- traiter les boucles implicites, pour accepter les entrées-sorties de Fortran de manière correcte ;
- étendre le sous ensemble accepté en entrée, autant en ce qui concerne le langage que les directives HPF, pour se rapprocher et atteindre le sous-ensemble officiel de HPF qui a été défini par la spécification ;
- explorer les diverses solutions envisageables et les mécanismes à prévoir pour permettre la compilation séparée de programmes HPF, et l'appel de routines avec des paramètres répartis ;
- élargir le champs d'application des optimisations, notamment de l'analyse des recouvrements.

Bibliographie

[ACIK93] Corinne Ancourt, Fabien Coelho, François Irigoin, and Ronan Keryell. Communication compilation for HPF. *Copies de transparents*, March 1993.

[ACP92] Françoise André, Olivier Chéron, and Jean-Louis Pazat. Compiling sequential programs for distributed memory parallel computers with Pandore II. Technical report, IRISA, May 1992. *13 références.*

[AI92] Corinne Ancourt and François Irigoin. Automatic code distribution. In *Third Workshop on Compilers for Parallel Computers*, July 1992. *13 références.*

[AL93a] Saman P. Amarasinghe and Monica S. Lam. Communication optimization and code generation for distributed memory machines. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1993. *28 références.*

[AL93b] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1993. *40 références.*

[Amd67] G. Amdalh. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conf.*, pages 483–485, 1967.

[Anc91] Corinne Ancourt. *Génération automatique de codes de transfert pour multiprocesseurs à mémoires locales.* PhD thesis, Université Paris VI, March 1991. *98 références.*

[Ani93] Laurent Aniort. Rapport de stage d'option, distribution sur un réseau de stations de travail d'un code de calcul de simulation de processus thermique. Rapport EMP-CRI A 239, CRI, École des mines de Paris, June 1993.

[ANS83] ANSI. *American National Standard Programming Language Fortran, ANSI x3.9-1978, ISO 1539-1980.* New-York, 1983.

[AP92] Françoise André and Thierry Priol. Programming distributed memory parallel computers whitout explicit message passing. In *Scalable High Performance Computing Conference*, pages 90–97, 1992. *14 références.*

[APT90] Françoise André, Jean-Louis Pazat, and Henry Thomas. Pandore: A system to manage data distribution. Publication Interne 519, IRISA, February 1990. *7 références.*

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[Baz91] Eric W. Bazin. Distribution d'une application scientifique sur un réseau de stations de travail, approche théorique et implémentation. Rapport EMP-CRI A 207, CRI, École des mines de Paris, September 1991.

[BBDS93] David Bailey, Eric Barszcz, Leonardo Dagum, and Horst Simon. NAS parallel benchmark results. *IEEE Parallel and Distributed Technology Systems and Applications*, pages 43–51, February 1993. *4 références.*

[BBLS91] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS parallel benchmarks. Report RNR-91 002, Numerical Aerodynamique Simulation Systems Division, NASA, August 1991. *38 références.*

[BCJT92] Cyrille Bareau, Benoît Caillaud, Claude Jard, and René Thoraval. Correctness of automated distribution of sequential programs. Rapport de Recherche 1724, INRIA, July 1992. *12 références.*

[BCZ92] Siegfried Benkner, Barbara M. Chapman, and Hans P. Zima. Vienna fortran 90. In *Scalable High Performance Computing Conference*, pages 51–59, 1992. *34 références.*

[BD93] Andy D. Ben-Dyke. Architectural taxonomy - a brief review. News, `comp.parallel`, June 1993. *5 références.*

[BK89] V. Balasundaram and Ken Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1989. *11 références.*

[Bra92a] Thomas Brandes. Adaptor language reference manual. Internal Report Adaptor 3, High Performance Computing Center, German National Research Institute for Computer Science, October 1992. ftp `ftp.gmd.de`, `gmd/adaptor`.

[Bra92b] Thomas Brandes. Adaptor the distributed array library. Internal Report Adaptor 4, High Performance Computing Center, German National Research Institute for Computer Science, October 1992. ftp `ftp.gmd.de`, `gmd/adaptor`.

[Bra92c] Thomas Brandes. Adaptor users guide. Internal Report Adaptor 2, High Performance Computing Center, German National Research Institute for Computer Science, October 1992. ftp `ftp.gmd.de`, `gmd/adaptor`.

[Bra92d] Thomas Brandes. Efficient data parallel programming without explicit message passing for distributed memory multiprocessors. Internal Report AHR-92 4, High Performance Computing Center, German National Research Institute for Computer Science, September 1992. *25 références*, ftp `ftp.gmd.de`, `gmd/adaptor`.

[Bra93a] Thomas Brandes. Adaptor: A compilation system for data parallel fortran programs. Technical report, High Performance Computing Center, German National Research Institute for Computer Science, August 1993. *22 références*, ftp `ftp.gmd.de`, `gmd/adaptor/docs`.

[Bra93b] Thomas Brandes. Results of adaptor with the Purdue Set. Internal Report AHR-93 3, High Performance Computing Center, German National Research Institute for Computer Science, August 1993. *1 références*, ftp `ftp.gmd.de`, `gmd/adaptor/docs`.

[Can92] David Cann. Retire Fortran? a debate relinked. *Communications of the ACM*, 35(8):81–89, August 1992. *24 références.*

[CC92] Marina Chen and James Cowie. Prototyping Fortran-90 compilers for Massively Parallel Machines. *ACM SIGPLAN Notices*, pages 94–105, June 1992. *7 références.*

[CGL+93] Siddhartha Chatterjee, John R. Gilbert, Fred J. E. Long, Robert Schreiber, and Shang-Hua Teng. Generating local addresses and communication sets for data-parallel programs. In *Symposium on Principles and Practice of Parallel Programming*, 1993. *10 références.*

[CK88a] D. Callahan and Ken Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Jounal of Parallel and Distributed Computing*, 5:517–550, 1988. *28 références.*

[CK88b] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988. *22 références.*

[CMZ92] Barbara Chapman, Piyush Mehrotra, and Hans Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992. *37 références.*

[CP93] P. Crooks and R. H. Perrott. Language constructs for data partitioning and distribution. *47 références*, soumis à Scientific Programming, 1993.

[Cyt86] Ron Cytron. Doacross: Beyond vectorization for multiprocessors (extended abstract). In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–844. IEEE, August 1986.

[DR93] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. RR 93 3, ENS-Lyon, January 1993. *21 références.*

[ESH93] Kalluri Eswar, P. Sadayappan, and Chua-Huang Huang. Compile-time characterisation of recurrent patterns in irregular computations. In *International Conference on Parallel Processing*, pages II–148–II–155, August 1993. *10 références.*

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part i, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[Fea92b] Paul Feautrier. Somme efficient solution to the affine scheduling problem, part ii, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.

[Fea93] Paul Feautrier. Toward automatic partitioning of arrays on distributed memory computers. In *International Conference on Supercomputing*, pages 175–184, July 1993. *25 références.*

[FGMS92] Jeanne Ferrante, G. R. Gao, Sam Midkiff, and Edith Schonberg. A critical survey of automatic data partinioning. Research report, IBM, 1992. *40 références.*

[Fly66] Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12), December 1966. *23 références.*

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, c-21(9):948–960, September 1972. *34 références.*

[For93] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Rice University, Houston, Texas, May 1993. *Version 1.0.*

[GBD+93] Al Geist, Adam Beguelin, Jack Dongarra, Jiang Weicheng, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.

[Ger89] Hans Michael Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Vienna, 1989. *66 références.*

[Ger91] Michael Gerndt. Work distribution in parallel programs for distributed memory multiprocessors. In *International Conference on Supercomputing*, pages 96–103, June 1991. *16 références.*

[Ger92] Michael Gerndt. Program analysis and transformations for message-passing programs. In *Scalable High Performance Computing Conference*, pages 60–67, 1992. *16 références.*

[GGM93] Chun Gong, Rajiv Gupta, and Rami Melhem. Compilation techniques for optimizing communication on distributed memory systems. In *International Conference on Parallel Processing*, pages II–39–II–46, August 1993. *16 références.*

[GKM+93] S. K. S. Gupta, S. D. Kaushik, S. Mufti, S. Sharma, C.-H. Huang, and P. Sadayappan. On compiling array expressions for efficient execution on distributed-memory machines. In *International Conference on Parallel Processing*, pages II–301–II–305, August 1993. *6 références.*

[GS92] Franco Gasperoni and Uwe Scheiegelshohn. Scheduling loop on parallel processors: A simple algorithm with close to optimum performance. Lecture Note, INRIA, 1992. *7 références.*

[GZ90] Hans Michael Gerndt and Hans Peter Zima. Optimizing communication in superb. In *CONPAR90*, pages 300–311, 1990. *19 références.*

[Hän82] Wolfgang Händer. Innovative computer architecture - how to increase parallelism but not complexity. In David J. Evans, editor, *Parallel Processing Systems - An Advanced Course*, pages 1–42. Cambridge University Press, 1982.

[Has93] Fawsi Hassaine. *Placement des données et Génération de code Automatique pour les Multiprocesseurs à Mémoire Distribuée.* PhD thesis, Université Paris VI, June 1993. *x références.*

[Hei93] Ernst A. Heinz. Modula-3*: An efficienly compilable extension of Modula-3 for problem-oriented explicitly parallel programming. In *Joint Symposium on Parallel Processing*, pages 269–276, May 1993. *13 références.*

[HK91] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991. *38 références.*

[HKT92a] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory machines. *Communications of the ACM*, 35(8):66–80, August 1992. *37 références.*

[HKT92b] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Evaluation of compiler optimizations for Fortran D on MIMD Distributed-Memory machines. In *International Conference on Supercomputing*, 1992. *38 références.*

[HLQA92] Philip J. Hatcher, Anthony J. Lapadula, Michael J. Quinn, and Ray J. Anderson. Compiling data-parallel programs for MIMD architectures. In *European Workshops on Parallel Computing*, 1992. *24 références.*

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990. *313 références.*

[IJT91] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the PIPS project. In *International Conference on Supercomputing*, June 1991. *37 références.*

[ISO91] ISO/IEC. *International Standard ISO/IEC 1539:1991 (E)*, second 1991-07-01 edition, 1991.

[Joh82] Eric E. Johnson. Completing an MIMD multiprocessor taxonomy. *ACM SIGARCH Computer Architecture Newsletter*, 16(3):44–47, June 1982. *13 références.*

[JT90] Pierre Jouvelot and Rémi Triolet. Newgen: A language-independent program generator. EMP-CRI 191, CRI, École des mines de Paris, December 1990. *11 références.*

[KF93] Alexander C. Klaiber and James L. Frankel. Comparing data-parallel and message-passing paradigms. In *International Conference on Parallel Processing*, pages II–11–II–20, August 1993. *12 références.*

[KLD92] Kathleen Knobe, Joan D. Lukas, and William J. Dally. Dynamic alignment on distributed memory systems. In *Third Workshop on Compilers for Parallel Computers*, pages 394–404, July 1992. *7 références.*

[KLSJ90] Kathleen Knobe, Joan D. Lukas, and Guy L. Steele Jr. Data optimization: Allocation of arrays to reduce communication on simd machines. *Jounal of Parallel and Distributed Computing*, 8, 1990.

[KM91] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991. *32 références.*

[KMVR90] Charles Koelbel, Piyush Mehrotra, and John Van Rosendale. Supporting shared data structures on distributed memory architectures. Technical Report ASD 915, Purdue University, January 1990. *9 références.*

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14:563–590, 1967.

[KN90] Kathleen Knobe and Venkataraman Natarajan. Data optimisation: Minimizing residual interprocessor data motion on SIMD machines. In *Third Symposium, on the Frontiers of Massively Parallel Computation*, pages 416–423, October 1990. *7 références.*

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, Inc., 1978.

[Lam74] Leslie Lamport. The parallel execution of do loops. *Communications of the ACM*, 17:83–93, February 1974.

[LC91] J. Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991. *20 références.*

[Lef93] Vincent Lefebvre. Répatition automatique des données en mémoire répartie. Copies de transparents, exposé de DEA-SI, Paris 6, *9 références*, March 1993.

[Ler93] Hugues Leroy. Les machines parallèles à mémoire distribuée. copies de transparents, journées GUS, June 1993. *2 références.*

[Lev92] John Levesque. *FORGE 90 and High Performance Fortran.* Applied Parallel Research, Inc., 1992. `xHPF77` presentation.

[Li86] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986. *102 références.*

[Lim93] Fujitsu Limited. Langage processors system on vpp500 system. Copies de transparents, confidentiel, 1993.

[Loi92] Michel Loi. Array redistribution on boolean cubes. In *Scalable High Performance Computing Conference*, 1992. *20 références.*

[LPT92] Oded Lempel, Shlomit S. Pinter, and Eli Turiel. Parallelizing a C dialect for Distributed Memory MIMD machines. In *Language and Compilers for Parallel Computing*, August 1992. *17 références.*

[Mer92] John Merlin. Techniques for the automatic parallelisation of 'Distributed Fortran 90'. SNARC 92 02, University of Southampton, 1992. *11 références.*

[MFvL+92] A. Gaber Mohamed, Geoffrey C. Fox, Gregor von Laszewski, Manish Parashar, Thomas Haupt, Kim Mills, Ying-Hua Lu, Neng-Tan Lin, and Nang-Kang Yeh. Application benchmark set for Fortran-D and high performance fortran. Technical Report 327, Syracuse University, Syracuse, New-York, 1992. *48 références.*

[MPM92] Tom MacDonald, Doug Pase, and Andy Meltzer. Addressing in Cray Research's MPP fortran. In *Third Workshop on Compilers for Parallel Computers*, pages 161–172, July 1992. *6 références.*

[MSS+88] Ravi Mirchandaney, Joel S. Saltz, Roger M. Smith, David M. Nicol, and Kay Crowley. Principles of runtime support for parallel processors. In *International Conference on Supercomputing*, pages 140–152, July 1988. *19 références.*

[OH92] Michael O'Boyle and G. A. Hedayat. Data alignment: Transformations to reduce communication on distributed memory architectures. In *Scalable High Performance Computing Conference*, pages 366–371, 1992. *12 références.*

[OR93] Joel Oren and Gowri Ramanathan. Survey of commercial parallel machines. News, `comp.parallel`, March 1993. *15 références.*

[Pas92] Douglas M. Pase. MPP fortran programming model. *7 références*, draft 1.2, January 1992.

[Paz93] Jean-Louis Pazat. Compilation de HPF. In *Journée Paradigme sur HPF*. École des mines de Paris, April 1993.

[PHL93] Michael Philippsen, Ernst A. Heinz, and Paul Lukowicz. Compiling machine-independent parallel programs. *ACM SIGPLAN Notices*, 28(8):99–108, August 1993. *22 références.*

[Pla92] Alexis Platonoff. Quel fortran pour la programmation massivement parallèle? Note CEA 2713, Commissariat à l'Énergie Atomique, November 1992. *16 références.*

[PSHG93] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling parallel programs for multiprocessors: Methodology and examples. *Computer*, pages 42–50, July 1993. *12 références.*

[RF76] S. S. Reddi and E. A. Feustel. A conceptual framework for computer architecture. *ACM Computing Surveys*, 8(2), June 1976. *33 références.*

[RF93] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, 7:9–50, 1993. *225 références.*

[RP89] Anne Rogers and Keshav Pingali. Process decomposition through locality of reference. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 1989. *18 références.*

[RP92] Anne Rogers and Keshav Pingali. Compiling for distributed memory architectures, June 1992. *35 références*, communication personnelle, sans doute un rapport technique.

[Sti93] James M. Stichnoth. Efficient compilation of array statements for private memory multicomputers. CMU-CS-93 109, School of Computer Science, Carnegie Mellon University, February 1993. *19 références.*

[Tse93] Chau-Wen Tseng. *An Optimising Fortran D Compiler for MIMD Distributed Memory Machines*. PhD thesis, Rice University, Houston, Texas, January 1993. *213 références.*

[WE93] Jian Wang and Christine Eisenbeis. Decomposed software pipelining: A new approach to exploit instruction level parallelism for loop programs. In *Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, January 1993. *18 références.*

[Wil93] Gregory Wilson. Timeline. News `comp.parallel`, article 4663, August 1993.

[ZBC+92] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna fortran - a language specification. ftp `cs.rice.edu` `public/HPFF/papers/vf.tex`, 1992. *45 références*, version 1.1.

[ZC90] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990. *227 références.*

[ZC93] Hans Zima and Barbara Mary Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, February 1993. *76 références.*

Annexe A

Fonctions de Support

La compilation du code fait appelle à de nombreuses fonctions de support qui assurent des différentes opérations nécessaires au cours de l'exécution. Ces fonctions ont été implémentées en Fortran 77, et utilisent la librairie d'échange de messages PVM 3 pour les communications entre processus.

A.1 rtrsfunctions.h

```
c
c function declarations
c
      external HPFC_OWNERP
      logical  HPFC_OWNERP
c
      external HPFC_SENDERP
      logical  HPFC_SENDERP
c
      external HPFC_COMPUTERP
      logical  HPFC_COMPUTERP
c
      external HPFC_COMPUTERINOWNERSP
      logical  HPFC_COMPUTERINOWNERSP
c
      external HPFC_LOCALIND
      integer  HPFC_LOCALIND
c
      external HPFC_LOCALINDGAMMA
      integer  HPFC_LOCALINDGAMMA
c
c that's all folks!
c
```

A.2 hpfc_param.h

```
c
c COMMON /HPFC'PARAM/
c
c the fellowing files has to be included:
c     include 'parameters.h'
c
      common /HPFC_PARAM/
```

```
     $      ATOT(NBOFARRAYS),
     $      TTOP(NBOFTEMPLATES),
     $      NODIMA(NBOFARRAYS),
     $      NODIMT(NBOFTEMPLATES),
     $      NODIMP(NBOFPROCESSORS),
     $      RANGEA(NBOFARRAYS, 7, 10),
     $      RANGET(NBOFTEMPLATES, 7, 3),
     $      RANGEP(NBOFPROCESSORS, 7, 3),
     $      ALIGN(NBOFARRAYS, 7, 3),
     $      DIST(NBOFTEMPLATES, 7, 2)
c
      integer
     $      ATOT, TTOP, NODIMA, NODIMT, NODIMP,
     $      RANGEA, RANGET, RANGEP, ALIGN, DIST
c
```

A.3 hpfc_procs.h

```
c
c COMMON /HPFC_PROCS/
c
c the fellowing files has to be included:
c     include 'parameters.h'
c
      common /HPFC_PROCS/
     $      MYLID,
     $      MYTID,
     $      HOSTTID,
     $      NODETIDS(MAXSIZEOFPROCS),
     $      MYPOS(7, NBOFPROCESSORS),
     $      NBOFTASKS,
     $      NODECHANNELS(MAXSIZEOFPROCS),
     $      HOSTCHANNEL,
     $      MCASTHOST,
     $      MCASTNODES
c
      integer
     $      MYLID, MYTID, HOSTTID, NODETIDS, NBOFTASKS, MYPOS,
     $      NODECHANNELS, HOSTCHANNEL,  MCASTHOST, MCASTNODES
c
```

A.4 hpfc_globs.h

```
c
c COMMON /HPFC_GLOBS/
c
c the fellowing files has to be included:
c     include 'parameters.h'
c
      common /HPFC_GLOBS/
     $      OWNERTID,
     $      OLID,
     $      OPN,
     $      OINDP(7),
     $      OREPLICATED,
     $      SENDERTID,
```

```
     $       SLID,
     $       COMPUTERTID,
     $       CLID,
     $       CPN,
     $       CINDP(7),
     $       CPOS(7),
     $       CPOSPN,
     $       CPOSCOMPUTED,
     $       NLID,
     $       NTID
c
      integer
     $       OWNERTID,     OLID, OPN, OINDP,
     $       SENDERTID,    SLID,
     $       COMPUTERTID, CLID, CPN, CINDP, CPOS, CPOSPN,
     $       NLID, NTID
c
      logical
     $       OREPLICATED, CPOSCOMPUTED
c
```

A.5 rtrsfunctions.f

```
c
c Runtime Support Functions for the
c
c High Performance Fortran Compiler
c
c-----------------
c
c HPFC_TEMPLATE
c
c this subroutine computes the coordonates of array number an
c on the template tn
c
      subroutine HPFC_TEMPLATE(inda, an, indt, tn)
      integer inda(7), an, indt(7), tn
      include 'parameters.h'
      include 'hpfc_param.h'
      integer i, adim
      do i=1, NODIMT(tn)
         adim = ALIGN(an, i, 1)
         if (adim.EQ.INTFLAG) then
c
c no alignment
c
            indt(i) = INTFLAG
         else
            if (adim.EQ.0) then
c
c constant alignment
c
               indt(i) = ALIGN(an, i, 3)
            else
c
c linear alignment
```

```
c
               indt(i) = (ALIGN(an, i, 2) * inda(adim))
     $                   + ALIGN(an, i, 3)
            endif
         endif
      enddo
      end
c
c-----------------
c
c HPFC_PROCESSORS
c
c this subroutine computes on which processor a template element is mapped
c
      subroutine HPFC_PROCESSORS(indt, tn, indp, pn, replicated)
      integer indt(7), tn, indp(7), pn
      logical replicated
      include 'parameters.h'
      include 'hpfc_param.h'
      integer i, tdim, param
      replicated = .FALSE.
      do i=1, NODIMP(pn)
         tdim = DIST(tn, i, 1)
c
c replicated if a non specified template dimension is distributed
c
         if (indt(tdim).EQ.INTFLAG) then
            replicated = .TRUE.
            indp(i) = INTFLAG
         else
            param = DIST(tn, i, 2)
            if (param.GT.0) then
c
c here the distribution is BLOCK
c
               indp(i) = (((indt(tdim) - RANGET(tn, tdim, 1)) / param)
     $                   + RANGEP(pn, i, 1))
            else
c
c here the distribution is CYCLIC
c
               param = - param
               indp(i) = ( (MOD((indt(tdim)
     $                   - RANGET(tn, tdim, 1)), (param*RANGEP(pn, i, 3)))
     $                   / param) + RANGEP(pn, i, 1) )
            endif
         endif
      enddo
      end
c
c-----------------
c
c HPFC_PROCLID
c
      subroutine HPFC_PROCLID(indp, pn, lid, replicated)
```

```
      integer indp(7), pn, lid
      logical replicated
      include 'parameters.h'
      include 'hpfc_param.h'
      include 'hpfc_procs.h'
      integer i, t
      if (replicated) then
         lid = INTFLAG
      else
         if (NODIMP(pn).EQ.0) then
            lid = 1
         else
            t = indp(1) - RANGEP(pn, 1, 1)
            do i=2, NODIMP(pn)
               t = (t * RANGEP(pn, i, 3))
     $                  + (indp(i) - RANGEP(pn, i, 1))
            enddo
            lid = t+1
         endif
      endif
      end
c
c-----------------
c
c HPFC_CMPPOS
c
      subroutine HPFC_CMPPOS(pn, lid, indp)
      integer pn, lid, indp(7)
      include 'parameters.h'
      include 'hpfc_param.h'
      integer ndim, i, t, dim, size
      ndim = NODIMP(pn)
      t = lid-1
      do i=0, ndim-1
         dim = ndim-i
         size = RANGEP(pn, dim, 3)
         indp(dim) = MOD(t, size) + RANGEP(pn, dim, 1)
         t = t / size
      enddo
      end
c
c-----------------
c
c HPFC_CMPOWNERS
c
      subroutine HPFC_CMPOWNERS(an, i1, i2, i3, i4, i5, i6, i7)
      integer an, i1, i2, i3, i4, i5, i6, i7
      include 'parameters.h'
      include 'hpfc_param.h'
      include 'hpfc_procs.h'
      include 'hpfc_globs.h'
      integer inda(7), indt(7), indp(7), tn, i
      inda(1) = i1
      inda(2) = i2
      inda(3) = i3
```

```
      inda(4) = i4
      inda(5) = i5
      inda(6) = i6
      inda(7) = i7
      tn  = ATOT(an)
      OPN = TTOP(tn)
      call HPFC_TEMPLATE(inda, an, indt, tn)
      call HPFC_PROCESSORS(indt, tn, OINDP, OPN, OREPLICATED)
      if (OREPLICATED) then
         OLID = INTFLAG
         OWNERTID = INTFLAG
         do i=1, NODIMP(OPN)
            if (OINDP(i).EQ.INTFLAG) then
               indp(i) = RANGEP(OPN, i, 1)
            else
               indp(i) = OINDP(i)
            endif
         enddo
         call HPFC_PROCLID(indp, OPN, SLID, .FALSE.)
         SENDERTID  = NODETIDS(SLID)
      else
         call HPFC_PROCLID(OINDP, OPN, OLID, OREPLICATED)
         OWNERTID  = NODETIDS(OLID)
         SENDERTID = OWNERTID
         SLID = OLID
      endif
      end
c
c-----------------
c
c HPFC_CMPCOMPUTER
c
      subroutine HPFC_CMPCOMPUTER(an, i1, i2, i3, i4, i5, i6, i7)
      integer an, i1, i2, i3, i4, i5, i6, i7
      include 'parameters.h'
      include 'hpfc_param.h'
      include 'hpfc_procs.h'
      include 'hpfc_globs.h'
      integer inda(7), indt(7), tn, i
      logical replicated
      inda(1) = i1
      inda(2) = i2
      inda(3) = i3
      inda(4) = i4
      inda(5) = i5
      inda(6) = i6
      inda(7) = i7
      tn  = ATOT(an)
      CPN = TTOP(tn)
      call HPFC_TEMPLATE(inda, an, indt, tn)
      call HPFC_PROCESSORS(indt, tn, CINDP, CPN, replicated)
      if (replicated) then
         do i=1, NODIMP(CPN)
            if (CINDP(i).EQ.INTFLAG)
     $              CINDP(i) = RANGEP(CPN, i, 1)
```

```
            enddo
            replicated = .FALSE.
         endif
         call HPFC_PROCLID(CINDP, CPN, CLID, replicated)
         COMPUTERTID  = NODETIDS(CLID)
         CPOSCOMPUTED = .FALSE.
         end
c
c-----------------
c
c HPFC_CMPNEIGHBOUR
c
c this function computes the neighbour tid of the current process,
c given the delta switch in the linearised processors representation.
c
         subroutine HPFC_CMPNEIGHBOUR(delta)
         integer delta
         include 'parameters.h'
         include 'hpfc_param.h'
         include 'hpfc_procs.h'
         include 'hpfc_globs.h'
         integer bufid
         NLID = MYLID + DELTA
         NTID = NODETIDS(NLID)
c
c where should I put it?
c
         call pvmfinitsend(PVMRAW, bufid)
         end
c
c-----------------
c
c HPFC_OWNERP
c
         logical function HPFC_OWNERP()
         include 'parameters.h'
         include 'hpfc_globs.h'
         include 'hpfc_procs.h'
         include 'hpfc_param.h'
         logical result
         integer i
         if (OREPLICATED) then
            result = .TRUE.
            do i=1, NODIMP(OPN)
               if (OINDP(i).NE.INTFLAG)
     $              result = result.AND.(OINDP(i).EQ.MYPOS(i, OPN))
            enddo
         else
            result = MYLID.EQ.OLID
         endif
         HPFC_OWNERP = result
         end
c
c-----------------
c
```

```
c HPFC_SENDERP
c
      logical function HPFC_SENDERP()
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      HPFC_SENDERP = MYLID.EQ.SLID
      end
c
c-----------------
c
c HPFC_COMPUTERP
c
      logical function HPFC_COMPUTERP()
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      HPFC_COMPUTERP = MYLID.EQ.CLID
      end
c
c-----------------
c
c HPFC_COMPUTERINOWNERSP
c
      logical function HPFC_COMPUTERINOWNERSP()
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      include 'hpfc_param.h'
      logical result
      integer i
      if (OREPLICATED) then
         result = .TRUE.
         if (OPN.EQ.CPN) then
c
c both owners and computer are set on the same processors.
c
            do i=1, NODIMP(OPN)
               if (OINDP(i).NE.INTFLAG)
     $              result = result.AND.(OINDP(i).EQ.CINDP(i))
            enddo
         else
c
c the processors are different, we have to compute the computer
c position on the owners' processors, if it hasn't been done.
c
            if ((.NOT.CPOSCOMPUTED).OR.(CPOSPN.NE.OPN)) then
               call HPFC_CMPPOS(OPN, CLID, CPOS)
               CPOSPN = OPN
               CPOSCOMPUTED = .TRUE.
            endif
            do i=1, NODIMP(OPN)
               if (OINDP(i).NE.INTFLAG)
     $              result = result.AND.(OINDP(i).EQ.CPOS(i))
            enddo
```

```
         endif
      else
         result = CLID.EQ.OLID
      endif
      HPFC_COMPUTERINOWNERSP = result
      end

c
c-----------------
c
c HPFC_SNDTO_C
c
c send to computer
c
      subroutine HPFC_SNDTO_C(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(what, val, 1, 1, info)
      call pvmfsend(COMPUTERTID, NODECHANNELS(CLID), info)
      NODECHANNELS(CLID) = NODECHANNELS(CLID) + 2
      end
c
c-----------------
c
c HPFC_SNDTO_H
c
c send to host
c
      subroutine HPFC_SNDTO_H(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(what, val, 1, 1, info)
      call pvmfsend(HOSTTID, HOSTCHANNEL, info)
      HOSTCHANNEL = HOSTCHANNEL + 2
      end
c
c-----------------
c
c HPFC_SNDTO_O
c
c send to owner, when there is just one
c
      subroutine HPFC_SNDTO_O(what, val)
      integer what
      include 'fpvm3.h'
```

```
      include 'parameters.h'
      include 'hpfc_procs.h'
      include 'hpfc_globs.h'
      integer bufid, info

      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(what, val, 1, 1, info)
      call pvmfsend(OWNERTID, NODECHANNELS(OLID), info)
      NODECHANNELS(OLID) = NODECHANNELS(OLID) + 2
      end
c
c-----------------
c
c HPFC_SNDTO_A
c
c send to all nodes
c
      subroutine HPFC_SNDTO_A(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(what, val, 1, 1, info)
      if (MYLID.GT.1)
     $     call pvmfmcast(
     $     MYLID-1,
     $     NODETIDS,
     $     MCASTNODES,
     $     info)
      if (MYLID.LT.NBOFTASKS)
     $     call pvmfmcast(
     $     NBOFTASKS-MYLID,
     $     NODETIDS(MYLID+1),
     $     MCASTNODES,
     $     info)
      MCASTNODES = MCASTNODES + 2
      end
c
c-----------------
c
c HPFC_HSNDTO_A
c
c host send to all nodes
c
      subroutine HPFC_HSNDTO_A(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
```

```
      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(what, val, 1, 1, info)
      call pvmfmcast(NBOFTASKS, NODETIDS, MCASTHOST, info)
      MCASTHOST = MCASTHOST + 2

      end
c
c-----------------
c
c HPFC_SNDTO_HA
c
c send to host and all nodes, but not the sender.
c
      subroutine HPFC_SNDTO_HA(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(what, val, 1, 1, info)
      if (MYLID.GT.1)
     $     call pvmfmcast(
     $     MYLID-1,
     $     NODETIDS,
     $     MCASTNODES,
     $     info)
      if (MYLID.LT.NBOFTASKS)
     $     call pvmfmcast(
     $     NBOFTASKS-MYLID,
     $     NODETIDS(MYLID+1),
     $     MCASTNODES,
     $     info)
      MCASTNODES = MCASTNODES + 2
      call pvmfsend(HOSTTID, HOSTCHANNEL, info)
      HOSTCHANNEL = HOSTCHANNEL + 2
      end
c
c-----------------
c
c HPFC_SNDTO_OS
c
c send to owners, by someone who is not in owners
c
      subroutine HPFC_SNDTO_OS(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_param.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info, i
      integer i1, i2, i3, i4, i5, i6, i7
      integer lb(7), ub(7), indp(7), lid
```

```
      if (OREPLICATED) then
         do i=1, NODIMP(OPN)

            if (OINDP(i).EQ.INTFLAG) then
               lb(i) = RANGEP(OPN, i, 1)
               ub(i) = RANGEP(OPN, i, 2)
            else
               lb(i) = OINDP(i)
               ub(i) = OINDP(i)
            endif

         enddo
         do i=(NODIMP(OPN)+1), 7
            lb(i) = 1
            ub(i) = 1
         enddo
         call pvmfinitsend(PVMRAW, bufid)
         call pvmfpack(what, val, 1, 1, info)
         do i7=lb(7), ub(7)
            indp(7) = i7
            do i6=lb(6), ub(6)
               indp(6) = i6
               do i5=lb(5), ub(5)
                  indp(5) = i5
                  do i4=lb(4), ub(4)
                     indp(4) = i4
                     do i3=lb(3), ub(3)
                        indp(3) = i3
                        do i2=lb(2), ub(2)
                           indp(2) = i2
                           do i1=lb(1), ub(1)
                              indp(1) = i1
c
c too much computation is made inside this call, and some
c factorisation could be made out of the inner loop, but I don't
c have time for that...
c
                              call HPFC_PROCLID(
     $                              indp, OPN, lid, .FALSE.)
                              call pvmfsend(
     $                              NODETIDS(lid),
     $                              NODECHANNELS(lid),
     $                              info)
                              NODECHANNELS(lid) = NODECHANNELS(lid) + 2

                           enddo
                        enddo
                     enddo
                  enddo
               enddo
            enddo
         enddo
      else
         call HPFC_SNDTO_O(what, val)
      endif
```

```
      end
c
c-----------------
c
c HPFC_SNDTO_OOS
c
c send to other owners, if any
c
      subroutine HPFC_SNDTO_OOS(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_param.h'
      include 'hpfc_procs.h'
      integer bufid, info,  i
      integer i1, i2, i3, i4, i5, i6, i7
      integer lb(7), ub(7), indp(7), lid
      do i=1, NODIMP(OPN)

         if (OINDP(i).EQ.INTFLAG) then
            lb(i) = RANGEP(OPN, i, 1)
            ub(i) = RANGEP(OPN, i, 2)
         else
            lb(i) = OINDP(i)
            ub(i) = OINDP(i)
         endif

      enddo
      do i=(NODIMP(OPN)+1), 7
         lb(i) = 1
         ub(i) = 1
      enddo
      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(what, val, 1, 1, info)
      do i7=lb(7), ub(7)
         indp(7) = i7
         do i6=lb(6), ub(6)
            indp(6) = i6
            do i5=lb(5), ub(5)
               indp(5) = i5
               do i4=lb(4), ub(4)
                  indp(4) = i4
                  do i3=lb(3), ub(3)
                     indp(3) = i3
                     do i2=lb(2), ub(2)
                        indp(2) = i2
                        do i1=lb(1), ub(1)
                           indp(1) = i1
c
c too much computation is made inside this call, and some
c factorisation could be made out of the inner loop, but I don't
c have time for that...
c
                           call HPFC_PROCLID(indp, OPN, lid, .FALSE.)
```

```
                          if (lid.NE.CLID) then
                             call pvmfsend(
     $                             NODETIDS(lid),
     $                             NODECHANNELS(lid),
     $                             info)
                             NODECHANNELS(lid) = NODECHANNELS(lid) + 2
                          endif
                       enddo
                    enddo
                 enddo
              enddo
           enddo
        enddo
      enddo
      end
c
c-----------------
c
c HPFC_SNDTO_NO
c
c send to not owners, when replicated...
c
      subroutine HPFC_SNDTO_NO(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_param.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info, i, indp(7), ndim
      logical result
      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(what, val, 1, 1, info)
      ndim = NODIMP(OPN)
      do lid=1, NBOFTASKS
c
c this leads to much much stupid computation, but all I need is the function
c done, and not the performances... however I think a better job could be done
c
c is lid in owners?
c
         call HPFC_CMPPOS(OPN, lid, indp)
         result = .TRUE.
         do i=1, ndim
            if (OINDP(i).NE.INTFLAG)
     $            result = result.AND.(OINDP(i).EQ.indp(i))
         enddo
         if (.NOT.result) then
            call pvmfsend(NODETIDS(lid), NODECHANNELS(lid), info)
            NODECHANNELS(lid) = NODECHANNELS(lid) + 2
         endif
      enddo
      end
c
c-----------------
```

```
c
c HPFC_SNDTO_HNO
c
c send to host and not owners, when replicated
c
      subroutine HPFC_SNDTO_HNO(what, val)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      call HPFC_SNDTO_H(what, val)
      call HPFC_SNDTO_NO(what, val)
      end
c
c-----------------
c
c HPFC_RCVFR_S
c
      subroutine HPFC_RCVFR_S(what, goal)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfrecv(SENDERTID, NODECHANNELS(SLID), bufid)
      call pvmfunpack(what, goal, 1, 1, info)
      NODECHANNELS(SLID) = NODECHANNELS(SLID) + 2
      end
c
c-----------------
c
c HPFC_RCVFR_C
c
      subroutine HPFC_RCVFR_C(what, goal)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfrecv(COMPUTERTID, NODECHANNELS(CLID), bufid)
      call pvmfunpack(what, goal, 1, 1, info)
      NODECHANNELS(CLID) = NODECHANNELS(CLID) + 2
      end
c
c-----------------
c
c HPFC_RCVFR_H
c
      subroutine HPFC_RCVFR_H(what, goal)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
```

```
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfrecv(HOSTTID, HOSTCHANNEL, bufid)
      call pvmfunpack(what, goal, 1, 1, info)
      HOSTCHANNEL = HOSTCHANNEL + 2

      end
c
c-----------------
c
c HPFC_RCVFR_mCS
c
      subroutine HPFC_RCVFR_mCS(what, goal)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfrecv(SENDERTID, MCASTNODES, bufid)
      call pvmfunpack(what, goal, 1, 1, info)
      MCASTNODES = MCASTNODES + 2
      end
c
c-----------------
c
c HPFC_RCVFR_mCH
c
      subroutine HPFC_RCVFR_mCH(what, goal)
      integer what
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer bufid, info


      call pvmfrecv(HOSTTID, MCASTHOST, bufid)
      call pvmfunpack(what, goal, 1, 1, info)
      MCASTHOST = MCASTHOST + 2

      end
c
c-----------------
c
c HPFC_LOCALIND
c
      integer function HPFC_LOCALIND(an, dim, indice)
      integer an, dim, indice
      include 'parameters.h'
      include 'hpfc_param.h'
c
c not implemented yet...
c
```

```
      HPFC_LOCALIND = 0
      end
c
c------------------
c
c HPFC_LOCALINDGAMMA
c
c computation of the gamma function for local indice access
c
      integer function HPFC_LOCALINDGAMMA(an, dim, indice)
      integer an, dim, indice
      include 'parameters.h'
      include 'hpfc_param.h'
      integer n, sc, no, bmtm
      integer t, nc, ni
c
c static parameters for the computation
c
      n    = RANGEA(an, dim, 5)
      sc   = RANGEA(an, dim, 6)
      no   = RANGEA(an, dim, 7)
      bmtm = RANGEA(an, dim, 8)
c
c the computation
c
      t    = indice + bmtm
      nc   = t / sc
      ni   = MOD(t, n)
      HPFC_LOCALINDGAMMA = n*(nc-no) + ni + 1
      end
c
c------------------
c
c HPFC_LOCALINDDELTA
c
c computation of the delta function for local indice access
c
      integer function HPFC_LOCALINDDELTA(an, dim, indice)
      integer an, dim, indice
      include 'parameters.h'
      include 'hpfc_param.h'
      integer n, sc, no, bmtm, rate, chsz
      integer t, nc, ni
c
c static parameters for the computation
c
      n    = RANGEA(an, dim, 5)
      sc   = RANGEA(an, dim, 6)
      no   = RANGEA(an, dim, 7)
      bmtm = RANGEA(an, dim, 8)
      rate = RANGEA(an, dim, 9)
      chsz = RANGEA(an, dim, 10)
c
c the computation
c
```

```
      t     = rate*indice + bmtm
      nc    = t / sc
      ni    = MOD(t, n) / abs(rate)
      HPFC_LOCALINDDELTA = chsz*(nc - no) + ni + 1
      end
c
c-----------------
c
c HPFC_INIT_NODE
c
      subroutine HPFC_INIT_NODE()
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_procs.h'
      integer i, bufid, info

      call pvmfmytid(MYTID)
      call pvmfparent(HOSTTID)
      NBOFTASKS = MAXSIZEOFPROCS

      call HPFC_INIT_COMMON_PROCS()
      call HPFC_INIT_COMMON_PARAM()

      call pvmfrecv(HOSTTID, 1, bufid)
      call pvmfunpack(INTEGER4, NODETIDS, NBOFTASKS, 1, info)
      do i=1, NBOFTASKS
         if (NODETIDS(i).EQ.MYTID)
     $           MYLID = i
      enddo
      do i=1, NBOFPROCESSORS
         call HPFC_CMPPOS(i, MYLID, MYPOS(1, i))
      enddo

      end
c
c-----------------
c
c HPFC_INIT_COMMON_PROCS
c
      subroutine HPFC_INIT_COMMON_PROCS()
      include 'parameters.h'
      include 'hpfc_procs.h'
      include 'hpfc_globs.h'
      integer i
      do i=1, NBOFTASKS
         NODECHANNELS(i) = 2
      enddo
      MCASTNODES = 1
      MCASTHOST  = 3
      OPN = 0
      CPOSPN = 0
      HOSTCHANNEL = 2
      end
c
c-----------------
```

```
c
c HPFC_INIT_COMMON_PARAM
c
c      subroutine HPFC_INIT_COMMON_PARAM()
c
c generated by the compiler...
c
c------------------
c
c HPFC_INIT_HOST
c
      subroutine HPFC_INIT_HOST()
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_procs.h'
      integer bufid, info
      call pvmfmytid(MYTID)
      MYLID = 0
      HOSTTID = MYTID
      NBOFTASKS = MAXSIZEOFPROCS

      call pvmfspawn(
     $       'node',
     $       PVMARCH,
     $       'SUN4',
     $       NBOFTASKS,
     $       NODETIDS,
     $       info)

      call pvmfinitsend(PVMRAW, bufid)
      call pvmfpack(INTEGER4, NODETIDS, NBOFTASKS, 1, info)
      call pvmfmcast(NBOFTASKS, NODETIDS, 1, info)

      call HPFC_INIT_COMMON_PROCS()
      call HPFC_INIT_COMMON_PARAM()

      end
c
c------------------
c
c HPFC_HOST_END
c
      subroutine HPFC_HOST_END()
      integer info
      call pvmfexit(info)
      end
c
c------------------
c
c HPFC_NODE_END
c
      subroutine HPFC_NODE_END()
      integer info
      call pvmfexit(info)
      end
```

```
c
c------------------
c
c HPFC_LOOP_BOUNDS
c
c Computation of the local loop bounds:
c nlb (new lower bound) and nub (new upper bound)
c and llb, the local lower bound in the former global loop,
c with lb (initial lower bound) and ub (initial upper bound)
c that access dimension of array da of array number an, mapped
c onto dimension of processors dp of processors number pn.
c
      subroutine HPFC_LOOP_BOUNDS(nlb, nub, llb, lb, ub, an, dp)
      integer nlb, nub, lb, ub, an, da, dp
      include 'parameters.h'
      include 'hpfc_param.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer d, n, tn, pn, b, dt, da
      tn = ATOT(an)
      pn = TTOP(tn)
      dt = DIST(tn, dp, 1)
      n  = DIST(tn, dp, 2)
      da = ALIGN(an, dt, 1)
      b  = ALIGN(an, dt, 3)
c
c d is the first template cell of the considered dimension that
c is mapped onto the given processors' cell:
c
      d  = n*(MYPOS(dp, pn)-RANGEP(pn, dp, 1)) +
     $     RANGET(tn, dt, 1)
      llb = MAX(lb, d-b)-1
      nlb = llb+b-d+2
      nub = MIN(ub+b, d+n-1)-d+1

      end
c
c------------------
c
c HPFC_PACK_1
c
      subroutine HPFC_PACK_1(
     $     what, array,
     $     dimlo1, dimup1,
     $     rglo1, rgup1, rgin1)
      integer
     $     what,
     $     dimlo1, dimup1,
     $     rglo1, rgup1, rgin1
      real
     $     array(dimlo1:dimup1)
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
```

```
      integer info
      integer i1
      do i1=rglo1, rgup1, rgin1
         call pvmfpack(what, array(i1), 1, 1, info)
      enddo
      end
c
c-----------------
c
c HPFC_PACK_2
c
      subroutine HPFC_PACK_2(
     $     what, array,
     $     dimlo1, dimup1,
     $     dimlo2, dimup2,
     $     rglo1, rgup1, rgin1,
     $     rglo2, rgup2, rgin2)
      integer
     $     what,
     $     dimlo1, dimup1,
     $     dimlo2, dimup2,
     $     rglo1, rgup1, rgin1,
     $     rglo2, rgup2, rgin2
      real
     $     array(dimlo1:dimup1, dimlo2:dimup2)
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      integer i1, i2
      do i1=rglo1, rgup1, rgin1
         do i2=rglo2, rgup2, rgin2
            call pvmfpack(what, array(i1, i2), 1, 1, info)
         enddo
      enddo
      end
c
c-----------------
c
c HPFC_PACK_3
c
      subroutine HPFC_PACK_3(
     $     what, array,
     $     dimlo1, dimup1,
     $     dimlo2, dimup2,
     $     dimlo3, dimup3,
     $     rglo1, rgup1, rgin1,
     $     rglo2, rgup2, rgin2,
     $     rglo3, rgup3, rgin3)
      integer
     $     what,
     $     dimlo1, dimup1,
     $     dimlo2, dimup2,
     $     dimlo3, dimup3,
```

```
     $       rglo1, rgup1, rgin1,
     $       rglo2, rgup2, rgin2,
     $       rglo3, rgup3, rgin3
      real
     $       array(dimlo1:dimup1, dimlo2:dimup2, dimlo3:dimup3)
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      integer i1, i2, i3
      do i1=rglo1, rgup1, rgin1
         do i2=rglo2, rgup2, rgin2
            do i3=rglo3, rgup3, rgin3
               call pvmfpack(what, array(i1, i2, i3), 1, 1, info)
            enddo
         enddo
      enddo
      end
c
c-----------------
c
c HPFC_PACK_4
c
      subroutine HPFC_PACK_4(
     $       what, array,
     $       dimlo1, dimup1,
     $       dimlo2, dimup2,
     $       dimlo3, dimup3,
     $       dimlo4, dimup4,
     $       rglo1, rgup1, rgin1,
     $       rglo2, rgup2, rgin2,
     $       rglo3, rgup3, rgin3,
     $       rglo4, rgup4, rgin4)
      integer
     $       what,
     $       dimlo1, dimup1,
     $       dimlo2, dimup2,
     $       dimlo3, dimup3,
     $       dimlo4, dimup4,
     $       rglo1, rgup1, rgin1,
     $       rglo2, rgup2, rgin2,
     $       rglo3, rgup3, rgin3,
     $       rglo4, rgup4, rgin4
      real
     $       array(dimlo1:dimup1, dimlo2:dimup2,
     $             dimlo3:dimup3, dimlo4:dimup4)
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      integer i1, i2, i3, i4
      do i1=rglo1, rgup1, rgin1
         do i2=rglo2, rgup2, rgin2
```

```
            do i3=rglo3, rgup3, rgin3
               do i4=rglo4, rgup4, rgin4
                  call pvmfpack(what,
     $                 array(i1, i2, i3, i4), 1, 1, info)
               enddo
            enddo
         enddo
      enddo
      end
c
c-----------------
c
c HPFC_UNPACK_1
c
      subroutine HPFC_UNPACK_1(
     $     what, array,
     $     dimlo1, dimup1,
     $     rglo1, rgup1, rgin1)
      integer
     $     what,
     $     dimlo1, dimup1,
     $     rglo1, rgup1, rgin1
      real
     $     array(dimlo1:dimup1)
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      integer i1
      do i1=rglo1, rgup1, rgin1
         call pvmfunpack(what, array(i1), 1, 1, info)
      enddo
      end
c
c-----------------
c
c HPFC_UNPACK_2
c
      subroutine HPFC_UNPACK_2(
     $     what, array,
     $     dimlo1, dimup1,
     $     dimlo2, dimup2,
     $     rglo1, rgup1, rgin1,
     $     rglo2, rgup2, rgin2)
      integer
     $     what,
     $     dimlo1, dimup1,
     $     dimlo2, dimup2,
     $     rglo1, rgup1, rgin1,
     $     rglo2, rgup2, rgin2
      real
     $     array(dimlo1:dimup1, dimlo2:dimup2)
      include 'fpvm3.h'
      include 'parameters.h'
```

```
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      integer i1, i2
      do i1=rglo1, rgup1, rgin1
         do i2=rglo2, rgup2, rgin2
            call pvmfunpack(what, array(i1, i2), 1, 1, info)
         enddo
      enddo
      end
c
c-----------------
c
c HPFC_UNPACK_3
c
      subroutine HPFC_UNPACK_3(
     $     what, array,
     $     dimlo1, dimup1,
     $     dimlo2, dimup2,
     $     dimlo3, dimup3,
     $     rglo1, rgup1, rgin1,
     $     rglo2, rgup2, rgin2,
     $     rglo3, rgup3, rgin3)
      integer
     $     what,
     $     dimlo1, dimup1,
     $     dimlo2, dimup2,
     $     dimlo3, dimup3,
     $     rglo1, rgup1, rgin1,
     $     rglo2, rgup2, rgin2,
     $     rglo3, rgup3, rgin3
      real
     $     array(dimlo1:dimup1, dimlo2:dimup2, dimlo3:dimup3)
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      integer i1, i2, i3
      do i1=rglo1, rgup1, rgin1
         do i2=rglo2, rgup2, rgin2
            do i3=rglo3, rgup3, rgin3
               call pvmfunpack(what, array(i1, i2, i3), 1, 1, info)
            enddo
         enddo
      enddo
      end
c
c-----------------
c
c HPFC_UNPACK_4
c
      subroutine HPFC_UNPACK_4(
     $     what, array,
     $     dimlo1, dimup1,
```

```
     $       dimlo2, dimup2,
     $       dimlo3, dimup3,
     $       dimlo4, dimup4,
     $       rglo1, rgup1, rgin1,
     $       rglo2, rgup2, rgin2,
     $       rglo3, rgup3, rgin3,
     $       rglo4, rgup4, rgin4)
      integer
     $       what,
     $       dimlo1, dimup1,
     $       dimlo2, dimup2,
     $       dimlo3, dimup3,
     $       dimlo4, dimup4,
     $       rglo1, rgup1, rgin1,
     $       rglo2, rgup2, rgin2,
     $       rglo3, rgup3, rgin3,
     $       rglo4, rgup4, rgin4
      real
     $       array(dimlo1:dimup1, dimlo2:dimup2,
     $             dimlo3:dimup3, dimlo4:dimup4)
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      integer i1, i2, i3, i4
      do i1=rglo1, rgup1, rgin1
         do i2=rglo2, rgup2, rgin2
            do i3=rglo3, rgup3, rgin3
               do i4=rglo4, rgup4, rgin4
                  call pvmfunpack(what,
     $                            array(i1, i2, i3, i4), 1, 1, info)
               enddo
            enddo
         enddo
      enddo
      end
c
c-----------------
c
c HPFC_RCVFR_N
c
      subroutine HPFC_RCVFR_N()
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      call pvmfrecv(NTID, NODECHANNELS(NLID), info)
      end
c
c-----------------
c
c HPFC_SNDTO_N
c
```

```
      subroutine HPFC_SNDTO_N()
      include 'fpvm3.h'
      include 'parameters.h'
      include 'hpfc_globs.h'
      include 'hpfc_procs.h'
      integer info
      call pvmfsend(NTID, NODECHANNELS(NLID), info)
      end
c
c------------------
```

Annexe B

Exemple

Voici pour exemple le code généré de manière automatique pour le programme `hpftest41`. 4 fichiers sont générés par le compilateur : le programme `host` pour la machine hôte, le programme `node` pour les nœuds, la routine `hpfc_init_common_param()`, qui initialise la description des données réparties, et `parameters.h` qui contient les différents paramètres de taille nécessaires.

B.1 host.f

```
c
c this file has been automatically generated by the hpf-compiler
c
      PROGRAM HOST
      INTEGER K,J,I,I_0,I_1,I_2,I_3
      REAL*4 F_4,F_3,F_2,F_1,F_0
      include 'fpvm3.h'
      include 'rtrsfunctions.h'
      include 'parameters.h'
      include 'hpfc_procs.h'
c initializes host and spawns nodes
      CALL HPFC_INIT_HOST
      WRITE (UNIT=6,FMT=*) 'HPFTEST41 RUNNING'
      WRITE (UNIT=6,FMT=*) 'THERMO'
C
C INITIALIZATION
      WRITE (UNIT=6,FMT=*) 'INITIALIZING'
C
C ITERATIONS... SHOULD BE A TEST, BUT REDUCTIONS ARE NOT ACCEPTED
      WRITE (UNIT=6,FMT=*) 'RUNNING'
C
C PRINT RESULTS
      WRITE (UNIT=6,FMT=*) 'RESULTS:'
10    FORMAT(F8.2,F8.2,F8.2,F8.2,F8.2)
      DO I = 2, 10, 2
         CALL HPFC_CMPOWNERS(3, I, 2, 0, 0, 0, 0, 0)
         CALL HPFC_RCVFR_S(REAL4, F_0)
         CALL HPFC_CMPOWNERS(3, I, 4, 0, 0, 0, 0, 0)
         CALL HPFC_RCVFR_S(REAL4, F_1)
         CALL HPFC_CMPOWNERS(3, I, 6, 0, 0, 0, 0, 0)
         CALL HPFC_RCVFR_S(REAL4, F_2)
```

```
          CALL HPFC_CMPOWNERS(3, I, 8, 0, 0, 0, 0, 0)
          CALL HPFC_RCVFR_S(REAL4, F_3)
          CALL HPFC_CMPOWNERS(3, I, 10, 0, 0, 0, 0, 0)
          CALL HPFC_RCVFR_S(REAL4, F_4)
          WRITE (FMT=10,UNIT=6) F_0,F_1,F_2,F_3,F_4
      ENDDO
      WRITE (UNIT=6,FMT=*) 'HPFTEST41 ENDED'
      CALL HPFC_HOST_END
      END
c
```

B.2 node.f

```
c
c this file has been automatically generated by the hpf-compiler
c
      PROGRAM NODE
      INTEGER I_45,I_44,I_43,I_42,I_41,I_40,I_39,I_38,I_37,I_36,I_35,
     &I_34,I_33,I_32,I_31,I_30,I_29,I_28,I_27,I_26,I_25,I_24,I_23,
     &I_22,I_21,I_20,I_19,I_18,I_17,I_16,I_15,I_14,I_13,I_12,I_11,
     &I_10,I_9,I_8,I_7,I_6,I_5,I_4,K,J,I,I_0,I_1,I_2,I_3
      REAL*4 F_4,F_3,F_2,F_1,F_0,NORTH(1:250),TS(1:250,1:250),TC(0:
     &251,0:251)
      include 'fpvm3.h'
      include 'rtrsfunctions.h'
      include 'parameters.h'
      include 'hpfc_procs.h'
      CALL HPFC_INIT_NODE
C
C INITIALIZATION
      IF (MYPOS(1,1).EQ.1) THEN
          CALL HPFC_LOOP_BOUNDS(I_5, I_6, I_7, 1, 500, 1, 2)
          DO I_4 = I_5, I_6
              NORTH(I_4) = 100.0
          ENDDO
      ENDIF
      IF (MYPOS(1,1).EQ.1) THEN
          CALL HPFC_LOOP_BOUNDS(I_9, I_10, I_11, 1, 500, 2, 2)
          DO I_8 = I_9, I_10
              TC(1,I_8) = NORTH(I_8)
              TS(1,I_8) = NORTH(I_8)
          ENDDO
      ENDIF
      CALL HPFC_LOOP_BOUNDS(I_13, I_14, I_15, 1, 500, 2, 2)
      CALL HPFC_LOOP_BOUNDS(I_17, I_18, I_19, 2, 500, 2, 1)
      DO I_12 = I_13, I_14
          DO I_16 = I_17, I_18
              TC(I_16,I_12) = 10.0
              TS(I_16,I_12) = 10.0
          ENDDO
      ENDDO
C
C ITERATIONS... SHOULD BE A TEST, BUT REDUCTIONS ARE NOT ACCEPTED
      DO K = 1, 100
c P(1:2, 2) send TC(1:250, 1) to (-1)
          IF (MYPOS(2,1).EQ.2) THEN
```

```
            CALL HPFC_CMPNEIGHBOUR(-1)
            CALL HPFC_PACK_2(REAL4, TC(TC_LO1,TC_LO2), TC_LO1, TC_UP1
     &       , TC_LO2, TC_UP2, 1, 250, 1, 1, 1, 1)
            CALL HPFC_SNDTO_N
         ENDIF
c P(1:2, 1) send TC(1:250, 250) to (+1)
         IF (MYPOS(2,1).EQ.1) THEN
            CALL HPFC_CMPNEIGHBOUR(1)
            CALL HPFC_PACK_2(REAL4, TC(TC_LO1,TC_LO2), TC_LO1, TC_UP1
     &       , TC_LO2, TC_UP2, 1, 250, 1, 250, 250, 1)
            CALL HPFC_SNDTO_N
         ENDIF
c P(2, 1:2) send TC(1, 1:250) to (-2)
         IF (MYPOS(1,1).EQ.2) THEN
            CALL HPFC_CMPNEIGHBOUR(-2)
            CALL HPFC_PACK_2(REAL4, TC(TC_LO1,TC_LO2), TC_LO1, TC_UP1
     &       , TC_LO2, TC_UP2, 1, 1, 1, 1, 250, 1)
            CALL HPFC_SNDTO_N
         ENDIF
c P(1, 1:2) send TC(250, 1:250) to (+2)
         IF (MYPOS(1,1).EQ.1) THEN
            CALL HPFC_CMPNEIGHBOUR(2)
            CALL HPFC_PACK_2(REAL4, TC(TC_LO1,TC_LO2), TC_LO1, TC_UP1
     &       , TC_LO2, TC_UP2, 250, 250, 1, 1, 250, 1)
            CALL HPFC_SNDTO_N
         ENDIF
c P(1:2, 1) receive TC(1:250, 251) from (+1)
         IF (MYPOS(2,1).EQ.1) THEN
            CALL HPFC_CMPNEIGHBOUR(1)
            CALL HPFC_RCVFR_N
            CALL HPFC_UNPACK_2(REAL4, TC(TC_LO1,TC_LO2), TC_LO1,
     &       TC_UP1, TC_LO2, TC_UP2, 1, 250, 1, 251, 251, 1)
         ENDIF
c P(1:2, 2) receive TC(1:250, 0) from (-1)
         IF (MYPOS(2,1).EQ.2) THEN
            CALL HPFC_CMPNEIGHBOUR(-1)
            CALL HPFC_RCVFR_N
            CALL HPFC_UNPACK_2(REAL4, TC(TC_LO1,TC_LO2), TC_LO1,
     &       TC_UP1, TC_LO2, TC_UP2, 1, 250, 1, 0, 0, 1)
         ENDIF
c P(1, 1:2) receive TC(251, 1:250) from (+2)
         IF (MYPOS(1,1).EQ.1) THEN
            CALL HPFC_CMPNEIGHBOUR(2)
            CALL HPFC_RCVFR_N
            CALL HPFC_UNPACK_2(REAL4, TC(TC_LO1,TC_LO2), TC_LO1,
     &       TC_UP1, TC_LO2, TC_UP2, 251, 251, 1, 1, 250, 1)
         ENDIF
c P(2, 1:2) receive TC(0, 1:250) from (-2)
         IF (MYPOS(1,1).EQ.2) THEN
            CALL HPFC_CMPNEIGHBOUR(-2)
            CALL HPFC_RCVFR_N
            CALL HPFC_UNPACK_2(REAL4, TC(TC_LO1,TC_LO2), TC_LO1,
     &       TC_UP1, TC_LO2, TC_UP2, 0, 0, 1, 1, 250, 1)
         ENDIF
         CALL HPFC_LOOP_BOUNDS(I_21, I_22, I_23, 2, 499, 2, 2)
```

```
      CALL HPFC_LOOP_BOUNDS(I_25, I_26, I_27, 2, 499, 2, 1)
      DO I_20 = I_21, I_22
         DO I_24 = I_25, I_26
            I_3 = J+1
            I_2 = J-1
            I_1 = I+1
            I_0 = I-1
            TS(I_24,I_20) = 0.25*(TC(I_24-1,I_20)+TC(I_24+1,I_20)+
     &         TC(I_24,I_20-1)+TC(I_24,I_20+1))
         ENDDO
      ENDDO
      CALL HPFC_LOOP_BOUNDS(I_29, I_30, I_31, 2, 499, 3, 2)
      CALL HPFC_LOOP_BOUNDS(I_33, I_34, I_35, 2, 499, 3, 1)
      DO I_28 = I_29, I_30
         DO I_32 = I_33, I_34
            TC(I_32,I_28) = TS(I_32,I_28)
         ENDDO
      ENDDO
   ENDDO
C
C PRINT RESULTS
   DO I = 2, 10, 2
      CALL HPFC_CMPOWNERS(3, I, 2, 0, 0, 0, 0, 0)
      IF (HPFC_SENDERP()) THEN
         I_36 = MOD(I-1, 250)+1
         I_37 = MOD(2-1, 250)+1
         CALL HPFC_SNDTO_H(REAL4, TC(I_36,I_37))
      ENDIF
      CALL HPFC_CMPOWNERS(3, I, 4, 0, 0, 0, 0, 0)
      IF (HPFC_SENDERP()) THEN
         I_38 = MOD(I-1, 250)+1
         I_39 = MOD(4-1, 250)+1
         CALL HPFC_SNDTO_H(REAL4, TC(I_38,I_39))
      ENDIF
      CALL HPFC_CMPOWNERS(3, I, 6, 0, 0, 0, 0, 0)
      IF (HPFC_SENDERP()) THEN
         I_40 = MOD(I-1, 250)+1
         I_41 = MOD(6-1, 250)+1
         CALL HPFC_SNDTO_H(REAL4, TC(I_40,I_41))
      ENDIF
      CALL HPFC_CMPOWNERS(3, I, 8, 0, 0, 0, 0, 0)
      IF (HPFC_SENDERP()) THEN
         I_42 = MOD(I-1, 250)+1
         I_43 = MOD(8-1, 250)+1
         CALL HPFC_SNDTO_H(REAL4, TC(I_42,I_43))
      ENDIF
      CALL HPFC_CMPOWNERS(3, I, 10, 0, 0, 0, 0, 0)
      IF (HPFC_SENDERP()) THEN
         I_44 = MOD(I-1, 250)+1
         I_45 = MOD(10-1, 250)+1
         CALL HPFC_SNDTO_H(REAL4, TC(I_44,I_45))
      ENDIF
   ENDDO
   CALL HPFC_NODE_END
   END
```

```
c
```

B.3 parameters.h

```
c
c file parameters.h
c
c this file has been automatically generated by the hpf-compiler
c
c type declarations
c
      integer
     $       INTFLAG,
     $       NBOFARRAYS,
     $       NBOFTEMPLATES,
     $       NBOFPROCESSORS,
     $       MAXSIZEOFPROCS

c
c parameters
c
      parameter(INTFLAG = -1234567890)
      parameter(NBOFARRAYS = 3)
      parameter(NBOFTEMPLATES = 1)
      parameter(NBOFPROCESSORS = 1)
      parameter(MAXSIZEOFPROCS = 4)
c
c dimensions
c
      integer
     $       NORTH_LO1, NORTH_UP1,
     $       TS_LO1, TS_UP1,
     $       TS_LO2, TS_UP2,
     $       TC_LO1, TC_UP1,
     $       TC_LO2, TC_UP2
c
c bounds values
c
      parameter(NORTH_LO1 = 1)
      parameter(NORTH_UP1 = 250)
      parameter(TS_LO1 = 1)
      parameter(TS_UP1 = 250)
      parameter(TS_LO2 = 1)
      parameter(TS_UP2 = 250)
      parameter(TC_LO1 = 0)
      parameter(TC_UP1 = 251)
      parameter(TC_LO2 = 0)
      parameter(TC_UP2 = 251)
c
c end of automatically generated file
c
```

B.4 init_param.f

```
c
c HPFC_INIT_COMMON_PARAM
```

```
c
c this file has been automatically generated by the hpf-compiler
c
      subroutine HPFC_INIT_COMMON_PARAM()
      include 'parameters.h'
      include 'hpfc_param.h'
c
c Arrays Initializations
c
c
c initializing array NORTH, number 1
c
      NODIMA(1) = 1
      ATOT(1) = 1

      RANGEA(1, 1, 1) = 1
      RANGEA(1, 1, 2) = 500
      RANGEA(1, 1, 3) = 500
c
      RANGEA(1, 1, 4) = 2
      RANGEA(1, 1, 5) = 250
      RANGEA(1, 1, 6) = 1
      RANGEA(1, 1, 7) = -1

      ALIGN(1, 1, 1) = 0
      ALIGN(1, 1, 2) = 0
      ALIGN(1, 1, 3) = 1
      ALIGN(1, 2, 1) = 1
      ALIGN(1, 2, 2) = 1
      ALIGN(1, 2, 3) = 0
c
c initializing array TS, number 2
c
      NODIMA(2) = 2
      ATOT(2) = 1

      RANGEA(2, 1, 1) = 1
      RANGEA(2, 1, 2) = 500
      RANGEA(2, 1, 3) = 500
c
      RANGEA(2, 1, 4) = 2
      RANGEA(2, 1, 5) = 250
      RANGEA(2, 1, 6) = 1
      RANGEA(2, 1, 7) = -1

      RANGEA(2, 2, 1) = 1
      RANGEA(2, 2, 2) = 500
      RANGEA(2, 2, 3) = 500
c
      RANGEA(2, 2, 4) = 2
      RANGEA(2, 2, 5) = 250
      RANGEA(2, 2, 6) = 1
      RANGEA(2, 2, 7) = -1

      ALIGN(2, 1, 1) = 1
```

```
      ALIGN(2, 1, 2) = 1
      ALIGN(2, 1, 3) = 0
      ALIGN(2, 2, 1) = 2
      ALIGN(2, 2, 2) = 1
      ALIGN(2, 2, 3) = 0
c
c initializing array TC, number 3
c
      NODIMA(3) = 2
      ATOT(3) = 1

      RANGEA(3, 1, 1) = 1
      RANGEA(3, 1, 2) = 500
      RANGEA(3, 1, 3) = 500
c
      RANGEA(3, 1, 4) = 2
      RANGEA(3, 1, 5) = 250
      RANGEA(3, 1, 6) = 1
      RANGEA(3, 1, 7) = -1

      RANGEA(3, 2, 1) = 1
      RANGEA(3, 2, 2) = 500
      RANGEA(3, 2, 3) = 500
c
      RANGEA(3, 2, 4) = 2
      RANGEA(3, 2, 5) = 250
      RANGEA(3, 2, 6) = 1
      RANGEA(3, 2, 7) = -1

      ALIGN(3, 1, 1) = 1
      ALIGN(3, 1, 2) = 1
      ALIGN(3, 1, 3) = 0
      ALIGN(3, 2, 1) = 2
      ALIGN(3, 2, 2) = 1
      ALIGN(3, 2, 3) = 0
c
c Templates Initializations
c
c
c initializing template T, number 1
c
      NODIMT(1) = 2
      TTOP(1) = 1

      RANGET(1, 1, 1) = 1
      RANGET(1, 1, 2) = 500
      RANGET(1, 1, 3) = 500

      RANGET(1, 2, 1) = 1
      RANGET(1, 2, 2) = 500
      RANGET(1, 2, 3) = 500

      DIST(1, 1, 1) = 1
      DIST(1, 1, 2) = 250
      DIST(1, 2, 1) = 2
```

```
      DIST(1, 2, 2) = 250
c
c Processors Initializations
c
c
c initializing processors P, number 1
c
      NODIMP(1) = 2

      RANGEP(1, 1, 1) = 1
      RANGEP(1, 1, 2) = 2
      RANGEP(1, 1, 3) = 2

      RANGEP(1, 2, 1) = 1
      RANGEP(1, 2, 2) = 2
      RANGEP(1, 2, 3) = 2
      end
c
c end of automatically generated fortran subroutine
c
```